

# Bcp

## 1.4

Generated by Doxygen 1.8.9.1

Thu Oct 8 2015 22:45:34



# Contents

<b>1</b>	<b>Hierarchical Index</b>	<b>1</b>
1.1	Class Hierarchy . . . . .	1
<b>2</b>	<b>Class Index</b>	<b>15</b>
2.1	Class List . . . . .	15
<b>3</b>	<b>File Index</b>	<b>19</b>
3.1	File List . . . . .	19
<b>4</b>	<b>Class Documentation</b>	<b>21</b>
4.1	BCP_buffer Class Reference . . . . .	21
4.1.1	Detailed Description . . . . .	23
4.1.2	Constructor & Destructor Documentation . . . . .	23
4.1.2.1	BCP_buffer . . . . .	23
4.1.2.2	BCP_buffer . . . . .	23
4.1.2.3	~BCP_buffer . . . . .	23
4.1.3	Member Function Documentation . . . . .	24
4.1.3.1	msgtag . . . . .	24
4.1.3.2	sender . . . . .	24
4.1.3.3	size . . . . .	24
4.1.3.4	data . . . . .	24
4.1.3.5	set_position . . . . .	24
4.1.3.6	set_size . . . . .	24
4.1.3.7	set_content . . . . .	24
4.1.3.8	operator= . . . . .	24
4.1.3.9	make_fit . . . . .	25
4.1.3.10	clear . . . . .	25
4.1.3.11	pack . . . . .	25
4.1.3.12	unpack . . . . .	25

4.1.3.13	pack	25
4.1.3.14	unpack	25
4.1.3.15	pack	25
4.1.3.16	pack	26
4.1.3.17	unpack	26
4.1.3.18	pack	26
4.1.3.19	pack	26
4.1.3.20	unpack	26
4.1.3.21	unpack	26
4.1.4	Member Data Documentation	26
4.1.4.1	_msgtag	26
4.1.4.2	_sender	26
4.1.4.3	_pos	27
4.1.4.4	_max_size	27
4.1.4.5	_size	27
4.1.4.6	_data	27
4.2	BCP_cg_par Struct Reference	27
4.3	BCP_cg_prob Class Reference	27
4.3.1	Detailed Description	29
4.3.2	Constructor & Destructor Documentation	29
4.3.2.1	BCP_cg_prob	29
4.3.2.2	~BCP_cg_prob	29
4.3.3	Member Function Documentation	29
4.3.3.1	has_ub	29
4.3.3.2	ub	29
4.3.3.3	probe_messages	29
4.3.3.4	unpack_var	29
4.3.4	Member Data Documentation	29
4.3.4.1	user	29
4.3.4.2	packer	30
4.3.4.3	msg_env	30
4.3.4.4	msg_buf	30
4.3.4.5	par	30
4.3.4.6	core	30
4.3.4.7	upper_bound	30
4.3.4.8	vars	30
4.3.4.9	x	31

4.3.4.10	sender	31
4.3.4.11	phase	31
4.3.4.12	node_level	31
4.3.4.13	node_index	31
4.3.4.14	node_iteration	31
4.4	BCP_cg_user Class Reference	31
4.4.1	Detailed Description	33
4.4.2	Constructor & Destructor Documentation	33
4.4.2.1	~BCP_cg_user	33
4.4.3	Member Function Documentation	33
4.4.3.1	send_cut	33
4.4.3.2	unpack_module_data	33
4.4.3.3	unpack_primal_solution	33
4.4.3.4	generate_cuts	34
4.5	BCP_col Class Reference	34
4.5.1	Detailed Description	35
4.5.2	Constructor & Destructor Documentation	35
4.5.2.1	BCP_col	35
4.5.2.2	BCP_col	36
4.5.2.3	BCP_col	36
4.5.2.4	BCP_col	36
4.5.2.5	~BCP_col	36
4.5.3	Member Function Documentation	36
4.5.3.1	Objective	36
4.5.3.2	LowerBound	36
4.5.3.3	UpperBound	36
4.5.3.4	Objective	36
4.5.3.5	LowerBound	37
4.5.3.6	UpperBound	37
4.5.3.7	operator=	37
4.5.3.8	assign	37
4.5.3.9	copy	37
4.5.3.10	copy	37
4.5.4	Member Data Documentation	37
4.5.4.1	_Objective	37
4.5.4.2	_LowerBound	38
4.5.4.3	_UpperBound	38

4.6	BCP_cut Class Reference	38
4.6.1	Detailed Description	39
4.6.2	Constructor & Destructor Documentation	40
4.6.2.1	BCP_cut	40
4.6.2.2	~BCP_cut	40
4.6.3	Member Function Documentation	40
4.6.3.1	obj_type	40
4.6.3.2	effective_count	40
4.6.3.3	lb	40
4.6.3.4	ub	40
4.6.3.5	bcpind	40
4.6.3.6	status	40
4.6.3.7	dont_send_to_pool	41
4.6.3.8	is_non_removable	41
4.6.3.9	is_to_be_removed	41
4.6.3.10	set_effective_count	41
4.6.3.11	increase_effective_count	41
4.6.3.12	decrease_effective_count	41
4.6.3.13	set_lb	41
4.6.3.14	set_ub	41
4.6.3.15	change_lb_ub_st	42
4.6.3.16	change_bounds	42
4.6.3.17	set_bcpind	42
4.6.3.18	set_status	42
4.6.3.19	dont_send_to_pool	42
4.6.3.20	make_active	42
4.6.3.21	make_non_removable	42
4.6.3.22	make_to_be_removed	42
4.6.4	Member Data Documentation	43
4.6.4.1	_lb	43
4.6.4.2	_ub	43
4.7	BCP_cut_algo Class Reference	43
4.7.1	Detailed Description	43
4.7.2	Constructor & Destructor Documentation	44
4.7.2.1	BCP_cut_algo	44
4.7.2.2	~BCP_cut_algo	44
4.7.3	Member Function Documentation	44

4.7.3.1	obj_type	44
4.8	BCP_cut_core Class Reference	44
4.8.1	Detailed Description	45
4.8.2	Constructor & Destructor Documentation	45
4.8.2.1	BCP_cut_core	45
4.8.2.2	BCP_cut_core	45
4.8.2.3	~BCP_cut_core	45
4.8.3	Member Function Documentation	45
4.8.3.1	obj_type	45
4.9	BCP_cut_set Class Reference	45
4.9.1	Detailed Description	46
4.9.2	Constructor & Destructor Documentation	46
4.9.2.1	BCP_cut_set	46
4.9.2.2	~BCP_cut_set	46
4.9.3	Member Function Documentation	47
4.9.3.1	append	47
4.9.3.2	append	47
4.9.3.3	set_lb_ub	47
4.9.3.4	set_lb_ub_st	47
4.9.3.5	set_lb_ub_st	47
4.9.3.6	move_deletable_to_pool	47
4.10	BCP_fatal_error Class Reference	47
4.10.1	Detailed Description	48
4.10.2	Constructor & Destructor Documentation	48
4.10.2.1	BCP_fatal_error	48
4.10.2.2	~BCP_fatal_error	48
4.11	BCP_internal_brobj Class Reference	48
4.11.1	Detailed Description	49
4.11.2	Constructor & Destructor Documentation	49
4.11.2.1	BCP_internal_brobj	49
4.11.2.2	BCP_internal_brobj	49
4.11.2.3	~BCP_internal_brobj	50
4.11.3	Member Function Documentation	50
4.11.3.1	child_num	50
4.11.3.2	affected_varnum	50
4.11.3.3	affected_cutnum	50
4.11.3.4	var_positions	50

4.11.3.5	cut_positions . . . . .	50
4.11.3.6	var_bounds_child . . . . .	50
4.11.3.7	cut_bounds_child . . . . .	50
4.11.3.8	apply_child_bounds . . . . .	51
4.11.3.9	pack . . . . .	51
4.11.3.10	unpack . . . . .	51
4.12	BCP_lp_branching_object Class Reference . . . . .	51
4.12.1	Detailed Description . . . . .	53
4.12.2	Constructor & Destructor Documentation . . . . .	53
4.12.2.1	BCP_lp_branching_object . . . . .	53
4.12.2.2	~BCP_lp_branching_object . . . . .	53
4.12.3	Member Function Documentation . . . . .	53
4.12.3.1	vars_affected . . . . .	53
4.12.3.2	cuts_affected . . . . .	54
4.12.3.3	vars_added . . . . .	54
4.12.3.4	cuts_added . . . . .	54
4.12.3.5	forced_var_bd_child . . . . .	54
4.12.3.6	forced_cut_bd_child . . . . .	54
4.12.3.7	implied_var_bd_child . . . . .	54
4.12.3.8	implied_cut_bd_child . . . . .	54
4.12.3.9	init_pos_for_added . . . . .	55
4.12.3.10	apply_child_bd . . . . .	55
4.12.3.11	print_branching_info . . . . .	55
4.12.4	Member Data Documentation . . . . .	55
4.12.4.1	child_num . . . . .	55
4.12.4.2	vars_to_add . . . . .	55
4.12.4.3	cuts_to_add . . . . .	55
4.12.4.4	forced_var_pos . . . . .	55
4.12.4.5	forced_cut_pos . . . . .	56
4.12.4.6	forced_var_bd . . . . .	56
4.12.4.7	forced_cut_bd . . . . .	56
4.13	BCP_lp_cut_pool Class Reference . . . . .	56
4.13.1	Detailed Description . . . . .	56
4.14	BCP_lp_integer_branching_object Class Reference . . . . .	56
4.14.1	Detailed Description . . . . .	57
4.15	BCP_lp_node Class Reference . . . . .	57
4.15.1	Detailed Description . . . . .	58



4.15.2	Member Data Documentation	58
4.15.2.1	user_data	58
4.16	BCP_lp_par Struct Reference	58
4.16.1	Detailed Description	59
4.16.2	Member Enumeration Documentation	59
4.16.2.1	chr_params	59
4.16.2.2	int_params	61
4.16.2.3	dbl_params	63
4.16.2.4	str_params	64
4.16.2.5	str_array_params	64
4.17	BCP_lp_parent Class Reference	65
4.17.1	Detailed Description	65
4.18	BCP_lp_prob Class Reference	65
4.18.1	Detailed Description	67
4.18.2	Member Data Documentation	67
4.18.2.1	packer	67
4.18.2.2	intAndSosObjects	67
4.18.2.3	node	68
4.18.2.4	parent	68
4.18.2.5	warmstartRoot	68
4.19	BCP_lp_relax Class Reference	68
4.19.1	Detailed Description	70
4.19.2	Constructor & Destructor Documentation	70
4.19.2.1	BCP_lp_relax	70
4.19.2.2	BCP_lp_relax	70
4.19.2.3	BCP_lp_relax	70
4.19.2.4	BCP_lp_relax	70
4.19.2.5	BCP_lp_relax	70
4.19.2.6	BCP_lp_relax	70
4.19.2.7	BCP_lp_relax	71
4.19.2.8	BCP_lp_relax	71
4.19.3	Member Function Documentation	71
4.19.3.1	colnum	71
4.19.3.2	rownum	71
4.19.3.3	Objective	71
4.19.3.4	ColLowerBound	71
4.19.3.5	ColUpperBound	71

4.19.3.6	<a href="#">RowLowerBound</a>	71
4.19.3.7	<a href="#">RowUpperBound</a>	72
4.19.3.8	<a href="#">operator=</a>	72
4.19.3.9	<a href="#">reserve</a>	72
4.19.3.10	<a href="#">clear</a>	72
4.19.3.11	<a href="#">erase_col_set</a>	72
4.19.3.12	<a href="#">erase_row_set</a>	72
4.19.3.13	<a href="#">pack</a>	72
4.19.3.14	<a href="#">unpack</a>	72
4.20	<a href="#">BCP_lp_result Class Reference</a>	72
4.20.1	<a href="#">Detailed Description</a>	73
4.20.2	<a href="#">Constructor &amp; Destructor Documentation</a>	73
4.20.2.1	<a href="#">BCP_lp_result</a>	73
4.20.2.2	<a href="#">~BCP_lp_result</a>	74
4.20.3	<a href="#">Member Function Documentation</a>	74
4.20.3.1	<a href="#">primalTolerance</a>	74
4.20.3.2	<a href="#">dualTolerance</a>	74
4.20.3.3	<a href="#">get_results</a>	74
4.20.3.4	<a href="#">fake_objective_value</a>	74
4.21	<a href="#">BCP_lp_sos_branching_object Class Reference</a>	74
4.21.1	<a href="#">Detailed Description</a>	74
4.22	<a href="#">BCP_lp_statistics Class Reference</a>	75
4.22.1	<a href="#">Detailed Description</a>	75
4.22.2	<a href="#">Member Function Documentation</a>	75
4.22.2.1	<a href="#">add</a>	75
4.23	<a href="#">BCP_lp_user Class Reference</a>	75
4.23.1	<a href="#">Detailed Description</a>	80
4.23.2	<a href="#">Constructor &amp; Destructor Documentation</a>	80
4.23.2.1	<a href="#">~BCP_lp_user</a>	80
4.23.3	<a href="#">Member Function Documentation</a>	81
4.23.3.1	<a href="#">over_ub</a>	81
4.23.3.2	<a href="#">select_nonzeros</a>	81
4.23.3.3	<a href="#">select_zeros</a>	81
4.23.3.4	<a href="#">select_positives</a>	81
4.23.3.5	<a href="#">select_fractions</a>	81
4.23.3.6	<a href="#">unpack_module_data</a>	81
4.23.3.7	<a href="#">process_message</a>	81

4.23.3.8	<a href="#">initialize_solver_interface</a>	81
4.23.3.9	<a href="#">initialize_int_and_sos_list</a>	82
4.23.3.10	<a href="#">initialize_new_search_tree_node</a>	82
4.23.3.11	<a href="#">load_problem</a>	82
4.23.3.12	<a href="#">modify_lp_parameters</a>	82
4.23.3.13	<a href="#">process_lp_result</a>	83
4.23.3.14	<a href="#">compute_lower_bound</a>	83
4.23.3.15	<a href="#">test_feasibility</a>	83
4.23.3.16	<a href="#">test_binary</a>	84
4.23.3.17	<a href="#">test_integral</a>	84
4.23.3.18	<a href="#">test_full</a>	84
4.23.3.19	<a href="#">generate_heuristic_solution</a>	84
4.23.3.20	<a href="#">pack_feasible_solution</a>	84
4.23.3.21	<a href="#">pack_primal_solution</a>	85
4.23.3.22	<a href="#">pack_dual_solution</a>	85
4.23.3.23	<a href="#">display_lp_solution</a>	85
4.23.3.24	<a href="#">restore_feasibility</a>	86
4.23.3.25	<a href="#">cuts_to_rows</a>	86
4.23.3.26	<a href="#">vars_to_cols</a>	87
4.23.3.27	<a href="#">generate_cuts_in_lp</a>	87
4.23.3.28	<a href="#">generate_vars_in_lp</a>	87
4.23.3.29	<a href="#">compare_cuts</a>	88
4.23.3.30	<a href="#">compare_vars</a>	88
4.23.3.31	<a href="#">logical_fixing</a>	88
4.23.3.32	<a href="#">reduced_cost_fixing</a>	89
4.23.3.33	<a href="#">select_branching_candidates</a>	89
4.23.3.34	<a href="#">branch_close_to_half</a>	89
4.23.3.35	<a href="#">branch_close_to_one</a>	90
4.23.3.36	<a href="#">append_branching_vars</a>	90
4.23.3.37	<a href="#">compare_branching_candidates</a>	90
4.23.3.38	<a href="#">set_actions_for_children</a>	90
4.23.3.39	<a href="#">set_user_data_for_children</a>	90
4.23.3.40	<a href="#">purge_slack_pool</a>	91
4.24	<a href="#">BCP_lp_var_pool Class Reference</a>	91
4.24.1	<a href="#">Detailed Description</a>	91
4.25	<a href="#">BCP_lp_waiting_col Class Reference</a>	91
4.25.1	<a href="#">Detailed Description</a>	91

4.26 BCP_lp_waiting_row Class Reference . . . . .	91
4.26.1 Detailed Description . . . . .	92
4.27 BCP_MemPool Class Reference . . . . .	92
4.27.1 Detailed Description . . . . .	92
4.28 BCP_message_environment Class Reference . . . . .	92
4.28.1 Detailed Description . . . . .	93
4.28.2 Constructor & Destructor Documentation . . . . .	93
4.28.2.1 ~BCP_message_environment . . . . .	93
4.28.3 Member Function Documentation . . . . .	94
4.28.3.1 register_process . . . . .	94
4.28.3.2 parent_process . . . . .	94
4.28.3.3 alive . . . . .	94
4.28.3.4 alive . . . . .	94
4.28.3.5 send . . . . .	94
4.28.3.6 send . . . . .	94
4.28.3.7 multicast . . . . .	94
4.28.3.8 multicast . . . . .	95
4.28.3.9 receive . . . . .	95
4.28.3.10 probe . . . . .	95
4.28.3.11 start_process . . . . .	95
4.28.3.12 start_process . . . . .	95
4.28.3.13 start_processes . . . . .	95
4.28.3.14 start_processes . . . . .	96
4.28.3.15 num_procs . . . . .	96
4.29 BCP_node_change Class Reference . . . . .	96
4.29.1 Detailed Description . . . . .	96
4.30 BCP_node_storage_in_tm Struct Reference . . . . .	96
4.30.1 Detailed Description . . . . .	97
4.31 BCP_obj_change Class Reference . . . . .	97
4.31.1 Detailed Description . . . . .	97
4.32 BCP_obj_set_change Class Reference . . . . .	97
4.32.1 Detailed Description . . . . .	97
4.33 BCP_parameter Class Reference . . . . .	97
4.33.1 Detailed Description . . . . .	98
4.33.2 Constructor & Destructor Documentation . . . . .	98
4.33.2.1 BCP_parameter . . . . .	98
4.33.2.2 BCP_parameter . . . . .	98

4.33.2.3	<a href="#">~BCP_parameter</a>	98
4.33.3	<a href="#">Member Function Documentation</a>	98
4.33.3.1	<a href="#">type</a>	98
4.33.3.2	<a href="#">index</a>	98
4.34	<a href="#">BCP_parameter_set&lt; Par &gt; Class Template Reference</a>	99
4.34.1	<a href="#">Detailed Description</a>	100
4.34.2	<a href="#">Constructor &amp; Destructor Documentation</a>	100
4.34.2.1	<a href="#">BCP_parameter_set</a>	100
4.34.2.2	<a href="#">~BCP_parameter_set</a>	101
4.34.3	<a href="#">Member Function Documentation</a>	101
4.34.3.1	<a href="#">read_from_stream</a>	101
4.34.3.2	<a href="#">read_from_file</a>	101
4.34.3.3	<a href="#">read_from_arglist</a>	101
4.34.3.4	<a href="#">write_to_stream</a>	101
4.34.3.5	<a href="#">pack</a>	101
4.34.3.6	<a href="#">unpack</a>	101
4.35	<a href="#">BCP_presolved_lp_brobj Class Reference</a>	102
4.35.1	<a href="#">Detailed Description</a>	103
4.35.2	<a href="#">Constructor &amp; Destructor Documentation</a>	103
4.35.2.1	<a href="#">BCP_presolved_lp_brobj</a>	103
4.35.2.2	<a href="#">~BCP_presolved_lp_brobj</a>	103
4.35.3	<a href="#">Member Function Documentation</a>	103
4.35.3.1	<a href="#">candidate</a>	103
4.35.3.2	<a href="#">candidate</a>	103
4.35.3.3	<a href="#">lpres</a>	104
4.35.3.4	<a href="#">action</a>	104
4.35.3.5	<a href="#">action</a>	104
4.35.3.6	<a href="#">user_data</a>	104
4.35.3.7	<a href="#">user_data</a>	104
4.35.3.8	<a href="#">fathomable</a>	104
4.35.3.9	<a href="#">had_numerical_problems</a>	104
4.35.3.10	<a href="#">get_lower_bounds</a>	104
4.35.3.11	<a href="#">set_lower_bounds</a>	105
4.35.3.12	<a href="#">get_results</a>	105
4.35.3.13	<a href="#">fake_objective_values</a>	105
4.35.3.14	<a href="#">set_objective_values</a>	105
4.36	<a href="#">BCP_problem_core Class Reference</a>	105

4.36.1	Detailed Description	106
4.36.2	Constructor & Destructor Documentation	106
4.36.2.1	BCP_problem_core	106
4.36.2.2	BCP_problem_core	106
4.36.2.3	~BCP_problem_core	107
4.36.3	Member Function Documentation	107
4.36.3.1	varnum	107
4.36.3.2	cutnum	107
4.36.3.3	pack	107
4.36.3.4	unpack	107
4.36.4	Member Data Documentation	107
4.36.4.1	vars	107
4.36.4.2	cuts	107
4.36.4.3	matrix	107
4.37	BCP_problem_core_change Class Reference	108
4.37.1	Detailed Description	109
4.37.2	Constructor & Destructor Documentation	109
4.37.2.1	BCP_problem_core_change	109
4.37.2.2	BCP_problem_core_change	110
4.37.2.3	BCP_problem_core_change	110
4.37.2.4	~BCP_problem_core_change	110
4.37.3	Member Function Documentation	110
4.37.3.1	storage	110
4.37.3.2	varnum	110
4.37.3.3	cutnum	110
4.37.3.4	operator=	110
4.37.3.5	ensure_explicit	110
4.37.3.6	make_wrtcore_if_shorter	111
4.37.3.7	swap	111
4.37.3.8	update	111
4.37.3.9	pack_size	111
4.37.3.10	pack	111
4.37.3.11	unpack	111
4.37.4	Member Data Documentation	111
4.37.4.1	_storage	111
4.37.4.2	var_pos	112
4.37.4.3	var_ch	112

4.37.4.4	cut_pos	112
4.37.4.5	cut_ch	112
4.38	BCP_process Class Reference	112
4.38.1	Detailed Description	112
4.39	BCP_row Class Reference	113
4.39.1	Detailed Description	114
4.39.2	Constructor & Destructor Documentation	114
4.39.2.1	BCP_row	114
4.39.2.2	BCP_row	114
4.39.2.3	BCP_row	114
4.39.2.4	BCP_row	114
4.39.2.5	~BCP_row	114
4.39.3	Member Function Documentation	115
4.39.3.1	LowerBound	115
4.39.3.2	UpperBound	115
4.39.3.3	LowerBound	115
4.39.3.4	UpperBound	115
4.39.3.5	operator=	115
4.39.3.6	assign	115
4.39.3.7	copy	115
4.39.3.8	copy	116
4.39.4	Member Data Documentation	116
4.39.4.1	_LowerBound	116
4.39.4.2	_UpperBound	116
4.40	BCP_scheduler Class Reference	116
4.40.1	Detailed Description	117
4.40.2	Constructor & Destructor Documentation	117
4.40.2.1	BCP_scheduler	117
4.40.3	Member Function Documentation	117
4.40.3.1	setParams	117
4.40.3.2	add_free_ids	117
4.40.3.3	request_sb_ids	118
4.40.3.4	release_sb_id	118
4.40.3.5	request_node_id	118
4.41	BCP_single_environment Class Reference	118
4.41.1	Detailed Description	119
4.41.2	Member Function Documentation	119

4.41.2.1	register_process	119
4.41.2.2	parent_process	119
4.41.2.3	alive	119
4.41.2.4	alive	120
4.41.2.5	send	120
4.41.2.6	send	120
4.41.2.7	multicast	120
4.41.2.8	multicast	120
4.41.2.9	receive	120
4.41.2.10	probe	120
4.41.2.11	start_process	121
4.41.2.12	start_process	121
4.41.2.13	start_processes	121
4.41.2.14	start_processes	121
4.42	BCP_slave_params Struct Reference	121
4.42.1	Detailed Description	122
4.43	BCP_solution Class Reference	122
4.43.1	Detailed Description	122
4.43.2	Constructor & Destructor Documentation	122
4.43.2.1	~BCP_solution	122
4.43.3	Member Function Documentation	122
4.43.3.1	objective_value	122
4.44	BCP_solution_generic Class Reference	123
4.44.1	Detailed Description	123
4.44.2	Constructor & Destructor Documentation	124
4.44.2.1	BCP_solution_generic	124
4.44.2.2	~BCP_solution_generic	124
4.44.3	Member Function Documentation	124
4.44.3.1	objective_value	124
4.44.3.2	set_objective_value	124
4.44.3.3	display	124
4.44.3.4	add_entry	124
4.44.4	Member Data Documentation	124
4.44.4.1	_objective	124
4.44.4.2	_delete_vars	125
4.44.4.3	_vars	125
4.44.4.4	_values	125



4.45 BCP_string Class Reference . . . . .	125
4.45.1 Detailed Description . . . . .	125
4.46 BCP_tm_flags Struct Reference . . . . .	125
4.46.1 Detailed Description . . . . .	126
4.46.2 Member Data Documentation . . . . .	126
4.46.2.1 root_pricing_unpacked . . . . .	126
4.47 BCP_tm_node Class Reference . . . . .	126
4.47.1 Detailed Description . . . . .	127
4.48 BCP_tm_node_data Class Reference . . . . .	127
4.48.1 Detailed Description . . . . .	127
4.49 BCP_tm_node_to_send Class Reference . . . . .	127
4.49.1 Detailed Description . . . . .	128
4.49.2 Member Function Documentation . . . . .	128
4.49.2.1 receive_node_desc . . . . .	128
4.49.2.2 receive_vars . . . . .	128
4.49.2.3 receive_cuts . . . . .	128
4.50 BCP_tm_par Struct Reference . . . . .	128
4.50.1 Detailed Description . . . . .	129
4.50.2 Member Enumeration Documentation . . . . .	129
4.50.2.1 chr_params . . . . .	129
4.50.2.2 int_params . . . . .	131
4.50.2.3 dbl_params . . . . .	131
4.50.2.4 str_params . . . . .	132
4.50.2.5 str_array_params . . . . .	132
4.51 BCP_tm_prob Class Reference . . . . .	132
4.51.1 Detailed Description . . . . .	134
4.51.2 Member Data Documentation . . . . .	134
4.51.2.1 packer . . . . .	134
4.51.2.2 root_node_sent_ . . . . .	134
4.51.2.3 lb_multiplier . . . . .	134
4.52 BCP_tm_stat Class Reference . . . . .	135
4.52.1 Detailed Description . . . . .	135
4.53 BCP_tm_user Class Reference . . . . .	135
4.53.1 Detailed Description . . . . .	137
4.53.2 Constructor & Destructor Documentation . . . . .	137
4.53.2.1 ~BCP_tm_user . . . . .	137
4.53.3 Member Function Documentation . . . . .	137

4.53.3.1	<a href="#">lower_bound</a>	137
4.53.3.2	<a href="#">pack_module_data</a>	137
4.53.3.3	<a href="#">unpack_feasible_solution</a>	137
4.53.3.4	<a href="#">replace_solution</a>	138
4.53.3.5	<a href="#">process_message</a>	138
4.53.3.6	<a href="#">initialize_core</a>	138
4.53.3.7	<a href="#">create_root</a>	138
4.53.3.8	<a href="#">display_node_information</a>	138
4.53.3.9	<a href="#">display_node_information</a>	138
4.53.3.10	<a href="#">display_final_information</a>	139
4.53.3.11	<a href="#">init_new_phase</a>	139
4.54	<a href="#">BCP_tree Class Reference</a>	139
4.54.1	<a href="#">Detailed Description</a>	139
4.55	<a href="#">BCP_ts_node_data Struct Reference</a>	140
4.55.1	<a href="#">Detailed Description</a>	140
4.56	<a href="#">BCP_ts_par Struct Reference</a>	140
4.56.1	<a href="#">Detailed Description</a>	140
4.56.2	<a href="#">Member Enumeration Documentation</a>	140
4.56.2.1	<a href="#">int_params</a>	140
4.57	<a href="#">BCP_ts_prob Class Reference</a>	140
4.57.1	<a href="#">Detailed Description</a>	141
4.58	<a href="#">BCP_ts_user Class Reference</a>	141
4.58.1	<a href="#">Detailed Description</a>	142
4.58.2	<a href="#">Constructor &amp; Destructor Documentation</a>	142
4.58.2.1	<a href="#">~BCP_ts_user</a>	142
4.58.3	<a href="#">Member Function Documentation</a>	142
4.58.3.1	<a href="#">unpack_module_data</a>	142
4.59	<a href="#">BCP_user_class Class Reference</a>	142
4.59.1	<a href="#">Detailed Description</a>	142
4.60	<a href="#">BCP_user_data Class Reference</a>	142
4.60.1	<a href="#">Detailed Description</a>	142
4.61	<a href="#">BCP_user_pack Class Reference</a>	143
4.61.1	<a href="#">Detailed Description</a>	143
4.61.2	<a href="#">Member Data Documentation</a>	143
4.61.2.1	<a href="#">user_class</a>	143
4.62	<a href="#">BCP_var Class Reference</a>	144
4.62.1	<a href="#">Detailed Description</a>	146

4.62.2	Constructor & Destructor Documentation	146
4.62.2.1	BCP_var	146
4.62.2.2	~BCP_var	146
4.62.3	Member Function Documentation	146
4.62.3.1	obj_type	146
4.62.3.2	var_type	146
4.62.3.3	obj	146
4.62.3.4	lb	146
4.62.3.5	ub	146
4.62.3.6	bcpind	147
4.62.3.7	status	147
4.62.3.8	dont_send_to_pool	147
4.62.3.9	is_fixed	147
4.62.3.10	is_fixed_to_zero	147
4.62.3.11	is_non_removable	147
4.62.3.12	is_removable	147
4.62.3.13	is_to_be_removed	147
4.62.3.14	set_var_type	148
4.62.3.15	set_obj	148
4.62.3.16	set_lb	148
4.62.3.17	set_ub	148
4.62.3.18	set_lb_ub	148
4.62.3.19	change_bounds	148
4.62.3.20	set_bcpind	148
4.62.3.21	set_status	148
4.62.3.22	dont_send_to_pool	149
4.62.3.23	make_active	149
4.62.3.24	make_non_removable	149
4.62.3.25	make_to_be_removed	149
4.62.3.26	display	149
4.62.4	Member Data Documentation	149
4.62.4.1	_var_type	149
4.62.4.2	_obj	149
4.62.4.3	_lb	149
4.62.4.4	_ub	150
4.63	BCP_var_algo Class Reference	150
4.63.1	Detailed Description	150

4.63.2	Constructor & Destructor Documentation	150
4.63.2.1	BCP_var_algo	150
4.63.2.2	~BCP_var_algo	151
4.63.3	Member Function Documentation	151
4.63.3.1	obj_type	151
4.64	BCP_var_core Class Reference	151
4.64.1	Detailed Description	151
4.64.2	Constructor & Destructor Documentation	152
4.64.2.1	BCP_var_core	152
4.64.2.2	BCP_var_core	152
4.64.2.3	~BCP_var_core	152
4.64.3	Member Function Documentation	152
4.64.3.1	obj_type	152
4.65	BCP_var_set Class Reference	152
4.65.1	Detailed Description	153
4.65.2	Constructor & Destructor Documentation	153
4.65.2.1	~BCP_var_set	153
4.65.3	Member Function Documentation	153
4.65.3.1	append	153
4.65.3.2	append	153
4.65.3.3	set_lb_ub	154
4.65.3.4	set_lb_ub_st	154
4.65.3.5	set_lb_ub_st	154
4.65.3.6	deletable	154
4.66	BCP_vec< T > Class Template Reference	154
4.66.1	Detailed Description	157
4.66.2	Constructor & Destructor Documentation	157
4.66.2.1	BCP_vec	157
4.66.2.2	BCP_vec	158
4.66.2.3	BCP_vec	158
4.66.2.4	BCP_vec	158
4.66.2.5	BCP_vec	158
4.66.2.6	~BCP_vec	158
4.66.3	Member Function Documentation	158
4.66.3.1	allocate	158
4.66.3.2	deallocate	158
4.66.3.3	insert_aux	158

4.66.3.4	begin	159
4.66.3.5	begin	159
4.66.3.6	end	159
4.66.3.7	end	159
4.66.3.8	entry	159
4.66.3.9	entry	159
4.66.3.10	index	159
4.66.3.11	size	159
4.66.3.12	capacity	159
4.66.3.13	empty	160
4.66.3.14	operator[]	160
4.66.3.15	operator[]	160
4.66.3.16	front	160
4.66.3.17	front	160
4.66.3.18	back	160
4.66.3.19	back	160
4.66.3.20	reserve	160
4.66.3.21	swap	160
4.66.3.22	operator=	161
4.66.3.23	assign	161
4.66.3.24	insert	161
4.66.3.25	insert	161
4.66.3.26	insert	161
4.66.3.27	insert	161
4.66.3.28	append	161
4.66.3.29	push_back	161
4.66.3.30	unchecked_push_back	162
4.66.3.31	pop_back	162
4.66.3.32	clear	162
4.66.3.33	update	162
4.66.3.34	unchecked_update	162
4.66.3.35	keep	162
4.66.3.36	keep	162
4.66.3.37	keep_by_index	162
4.66.3.38	unchecked_keep_by_index	163
4.66.3.39	keep_by_index	163
4.66.3.40	unchecked_keep_by_index	163

4.66.3.41 erase	163
4.66.3.42 erase	163
4.66.3.43 erase_by_index	163
4.66.3.44 unchecked_erase_by_index	163
4.66.3.45 erase_by_index	163
4.66.3.46 unchecked_erase_by_index	164
4.66.4 Member Data Documentation	164
4.66.4.1 start	164
4.66.4.2 finish	164
4.66.4.3 end_of_storage	164
4.67 BCP_vec_change< T > Class Template Reference	164
4.67.1 Detailed Description	165
4.67.2 Constructor & Destructor Documentation	165
4.67.2.1 BCP_vec_change	165
4.67.2.2 BCP_vec_change	165
4.67.2.3 BCP_vec_change	166
4.67.2.4 BCP_vec_change	167
4.67.2.5 BCP_vec_change	167
4.67.2.6 ~BCP_vec_change	167
4.67.3 Member Function Documentation	167
4.67.3.1 storage_size	167
4.67.3.2 update	167
4.67.3.3 pack	167
4.67.3.4 unpack	168
4.68 BCP_vg_par Struct Reference	168
4.68.1 Detailed Description	168
4.68.2 Member Enumeration Documentation	168
4.68.2.1 chr_params	168
4.68.2.2 int_params	168
4.69 BCP_vg_prob Class Reference	169
4.69.1 Detailed Description	170
4.69.2 Constructor & Destructor Documentation	170
4.69.2.1 BCP_vg_prob	170
4.69.2.2 ~BCP_vg_prob	170
4.69.3 Member Function Documentation	170
4.69.3.1 has_ub	170
4.69.3.2 ub	171

4.69.3.3	probe_messages	171
4.69.3.4	unpack_cut	171
4.69.4	Member Data Documentation	171
4.69.4.1	user	171
4.69.4.2	packer	171
4.69.4.3	msg_env	171
4.69.4.4	msg_buf	171
4.69.4.5	par	171
4.69.4.6	core	172
4.69.4.7	upper_bound	172
4.69.4.8	cuts	172
4.69.4.9	pi	172
4.69.4.10	sender	172
4.69.4.11	phase	172
4.69.4.12	node_level	172
4.69.4.13	node_index	172
4.69.4.14	node_iteration	173
4.70	BCP_vg_user Class Reference	173
4.70.1	Detailed Description	174
4.70.2	Constructor & Destructor Documentation	174
4.70.2.1	~BCP_vg_user	174
4.70.3	Member Function Documentation	175
4.70.3.1	send_var	175
4.70.3.2	unpack_module_data	175
4.70.3.3	unpack_dual_solution	175
4.70.3.4	generate_vars	175
4.71	BCP_warmstart Class Reference	175
4.71.1	Detailed Description	176
4.71.2	Constructor & Destructor Documentation	176
4.71.2.1	~BCP_warmstart	176
4.71.3	Member Function Documentation	176
4.71.3.1	storage	176
4.71.3.2	update	176
4.71.3.3	as_change	177
4.71.3.4	clone	177
4.71.3.5	empty_wrt_this	177
4.71.3.6	storage_size	177

4.72 BCP_warmstart_basis Class Reference . . . . .	177
4.72.1 Detailed Description . . . . .	178
4.72.2 Member Function Documentation . . . . .	178
4.72.2.1 convert_to_CoinWarmStart . . . . .	178
4.72.2.2 storage . . . . .	179
4.72.2.3 clone . . . . .	179
4.72.2.4 empty_wrt_this . . . . .	179
4.72.2.5 storage_size . . . . .	179
4.72.2.6 update . . . . .	179
4.72.2.7 as_change . . . . .	179
4.72.2.8 pack . . . . .	180
4.73 BCP_warmstart_dual Class Reference . . . . .	180
4.73.1 Detailed Description . . . . .	181
4.73.2 Constructor & Destructor Documentation . . . . .	181
4.73.2.1 BCP_warmstart_dual . . . . .	181
4.73.3 Member Function Documentation . . . . .	181
4.73.3.1 convert_to_CoinWarmStart . . . . .	181
4.73.3.2 storage . . . . .	181
4.73.3.3 clone . . . . .	181
4.73.3.4 empty_wrt_this . . . . .	181
4.73.3.5 storage_size . . . . .	182
4.73.3.6 update . . . . .	182
4.73.3.7 as_change . . . . .	182
4.73.3.8 pack . . . . .	182
4.74 BCP_warmstart_primaldual Class Reference . . . . .	183
4.74.1 Detailed Description . . . . .	183
4.74.2 Constructor & Destructor Documentation . . . . .	184
4.74.2.1 BCP_warmstart_primaldual . . . . .	184
4.74.3 Member Function Documentation . . . . .	184
4.74.3.1 convert_to_CoinWarmStart . . . . .	184
4.74.3.2 storage . . . . .	184
4.74.3.3 clone . . . . .	184
4.74.3.4 empty_wrt_this . . . . .	184
4.74.3.5 storage_size . . . . .	184
4.74.3.6 update . . . . .	185
4.74.3.7 as_change . . . . .	185
4.74.3.8 pack . . . . .	185



---

4.75 USER_initialize Class Reference . . . . .	185
4.75.1 Detailed Description . . . . .	186
4.75.2 Member Function Documentation . . . . .	186
4.75.2.1 msgenv_init . . . . .	186
<b>Index</b>	<b>187</b>



# Chapter 1

## Hierarchical Index

### 1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

<code>_EKKfactinfo</code>	
<code>AbcDualRowPivot</code>	
<code>AbcDualRowDantzig</code>	
<code>AbcDualRowSteepest</code>	
<code>AbcMatrix</code>	
<code>AbcMatrix2</code>	
<code>AbcMatrix3</code>	
<code>AbcNonLinearCost</code>	
<code>AbcPrimalColumnPivot</code>	
<code>AbcPrimalColumnDantzig</code>	
<code>AbcPrimalColumnSteepest</code>	
<code>AbcSimplexFactorization</code>	
<code>AbcTolerancesEtc</code>	
<code>AbcWarmStartOrganizer</code>	
<code>forcing_constraint_action</code>	
<code>tripleton_action</code>	
<code>doubleton_action</code>	
<code>remove_fixed_action</code>	
<code>std::allocator&lt; T &gt;</code>	
<code>ampl_info</code>	
<code>OsiSolverInterface::ApplyCutsReturnCode</code>	
<code>std::array&lt; T &gt;</code>	
<code>std::auto_ptr&lt; T &gt;</code>	
<code>auxiliary_graph</code>	
<code>std::basic_string&lt; Char &gt;</code>	
<code>std::string</code>	
<code>std::wstring</code>	
<code>std::basic_string&lt; char &gt;</code>	
<code>std::basic_string&lt; wchar_t &gt;</code>	
<code>BCP_buffer</code>	21
<code>BCP_cg_par</code>	27
<code>BCP_parameter_set&lt; BCP_cg_par &gt;</code>	99
<code>BCP_fatal_error</code>	47

BCP_internal_brobj . . . . .	48
BCP_lp_branching_object . . . . .	51
BCP_lp_node . . . . .	57
BCP_lp_par . . . . .	58
BCP_parameter_set< BCP_lp_par > . . . . .	99
BCP_lp_parent . . . . .	65
BCP_lp_result . . . . .	72
BCP_lp_statistics . . . . .	75
BCP_lp_waiting_col . . . . .	91
BCP_lp_waiting_row . . . . .	91
BCP_MemPool . . . . .	92
BCP_message_environment . . . . .	92
BCP_single_environment . . . . .	118
BCP_node_storage_in_tm . . . . .	96
BCP_obj_change . . . . .	97
BCP_obj_set_change . . . . .	97
BCP_parameter . . . . .	97
BCP_presolved_lp_brobj . . . . .	102
BCP_problem_core . . . . .	105
BCP_problem_core_change . . . . .	108
BCP_process . . . . .	112
BCP_cg_prob . . . . .	27
BCP_lp_prob . . . . .	65
BCP_tm_prob . . . . .	132
BCP_ts_prob . . . . .	140
BCP_vg_prob . . . . .	169
BCP_scheduler . . . . .	116
BCP_slave_params . . . . .	121
BCP_solution . . . . .	122
BCP_solution_generic . . . . .	123
BCP_string . . . . .	125
BCP_tm_flags . . . . .	125
BCP_tm_node_data . . . . .	127
BCP_tm_node_to_send . . . . .	127
BCP_tm_par . . . . .	128
BCP_parameter_set< BCP_tm_par > . . . . .	99
BCP_tm_stat . . . . .	135
BCP_tree . . . . .	139
BCP_ts_node_data . . . . .	140
BCP_ts_par . . . . .	140
BCP_parameter_set< BCP_ts_par > . . . . .	99
BCP_user_class . . . . .	142
BCP_cg_user . . . . .	31
BCP_lp_user . . . . .	75
BCP_tm_user . . . . .	135
BCP_ts_user . . . . .	141
BCP_vg_user . . . . .	173
BCP_user_pack . . . . .	143
BCP_vec< T > . . . . .	154
BCP_vec< BCP_child_action > . . . . .	154
BCP_vec< BCP_col * > . . . . .	154
BCP_vec< BCP_cut * > . . . . .	154

BCP_cut_set	45
BCP_vec< BCP_cut_core * >	154
BCP_vec< BCP_lp_result * >	154
BCP_vec< BCP_lp_waiting_col * >	154
BCP_lp_var_pool	91
BCP_vec< BCP_lp_waiting_row * >	154
BCP_lp_cut_pool	56
BCP_vec< BCP_obj_change >	154
BCP_vec< BCP_row * >	154
BCP_vec< BCP_string >	154
BCP_vec< BCP_tm_node * >	154
BCP_vec< BCP_user_data * >	154
BCP_vec< BCP_var * >	154
BCP_var_set	152
BCP_vec< BCP_var_core * >	154
BCP_vec< BCP_vec< BCP_cut * > >	154
BCP_vec< BCP_vec< BCP_row * > >	154
BCP_vec< char >	154
BCP_vec< Coin::SmartPtr< BCP_cut > >	154
BCP_vec< Coin::SmartPtr< BCP_var > >	154
BCP_vec< double >	154
BCP_vec< int >	154
BCP_vec< std::pair< BCP_string, BCP_parameter > >	154
BCP_vec< std::pair< int, int > >	154
BCP_vec_change< T >	164
BCP_vec_change< char >	164
BCP_vec_change< double >	164
BCP_vg_par	168
BCP_parameter_set< BCP_vg_par >	99
BCP_warmstart	175
BCP_warmstart_basis	177
BCP_warmstart_dual	180
BCP_warmstart_primaldual	183
std::bitset< Bits >	
BitVector128[external]	
blockStruct[external]	
blockStruct3[external]	
ClpNode::branchState[external]	
CbcOrClpParam[external]	
Cgl012Cut[external]	
cgl_arc[external]	
cgl_graph[external]	
cgl_node[external]	
CglBK[external]	
CglCutGenerator[external]	
CglAllDifferent[external]	
CglClique[external]	
CglFakeClique[external]	
CglDuplicateRow[external]	
CglFlowCover[external]	
CglGMI[external]	
CglGomory[external]	
CglImplication[external]	

- CglKnapsackCover [external]
- CglLandP [external]
- CglLiftAndProject [external]
- CglMixedIntegerRounding [external]
- CglMixedIntegerRounding2 [external]
- CglOddHole [external]
- CglProbing [external]
- CglRedSplit [external]
- CglRedSplit2 [external]
- CglResidualCapacity [external]
- CglSimpleRounding [external]
- CglStored [external]
- CglTwomir [external]
- CglZeroHalf [external]
- CglFlowVUB [external]
- CglHashLink [external]
- LAP::CglLandPSimplex [external]
- CglMixIntRoundVUB [external]
- CglMixIntRoundVUB2 [external]
- CglParam [external]
  - CglGMIParam [external]
  - CglLandP::Parameters [external]
  - CglRedSplit2Param [external]
  - CglRedSplitParam [external]
- CglPreProcess [external]
- CglTreeInfo [external]
  - CglTreeProbingInfo [external]
- CglUniqueRowCuts [external]
- CliqueEntry [external]
- CglProbing::CliqueType [external]
- ClpCholeskyBase [external]
  - ClpCholeskyDense [external]
  - ClpCholeskyMumps [external]
  - ClpCholeskyTaucs [external]
  - ClpCholeskyUfl [external]
  - ClpCholeskyWssmp [external]
  - ClpCholeskyWssmpKKT [external]
- ClpCholeskyDenseC [external]
- ClpConstraint [external]
  - ClpConstraintLinear [external]
  - ClpConstraintQuadratic [external]
- ClpDataSave [external]
- ClpDisasterHandler [external]
  - OsiClpDisasterHandler [external]
- ClpDualRowPivot [external]
  - ClpDualRowDantzig [external]
  - ClpDualRowSteepest [external]
- ClpEventHandler [external]
  - MyEventHandler [external]
- ClpFactorization [external]
- ClpHashValue [external]
- ClpLsqr [external]
- ClpMatrixBase [external]
  - ClpDummyMatrix [external]

- ClpNetworkMatrix [external]
- ClpPackedMatrix [external]
  - ClpDynamicMatrix [external]
  - ClpDynamicExampleMatrix [external]
  - ClpGubMatrix [external]
  - ClpGubDynamicMatrix [external]
- ClpPlusMinusOneMatrix [external]
- ClpModel [external]
  - ClpInterior [external]
    - ClpPdco [external]
    - ClpPredictorCorrector [external]
  - ClpSimplex [external]
    - AbcSimplex [external]
      - AbcSimplexDual [external]
      - AbcSimplexPrimal [external]
    - ClpSimplexDual [external]
    - ClpSimplexOther [external]
    - ClpSimplexPrimal [external]
    - ClpSimplexNonlinear [external]
- ClpNetworkBasis [external]
- ClpNode [external]
- ClpNodeStuff [external]
- ClpNonLinearCost [external]
- ClpObjective [external]
  - ClpLinearObjective [external]
  - ClpQuadraticObjective [external]
- ClpPackedMatrix2 [external]
- ClpPackedMatrix3 [external]
- ClpPdcoBase [external]
- ClpPresolve [external]
- ClpPrimalColumnPivot [external]
  - ClpPrimalColumnDantzig [external]
  - ClpPrimalColumnSteepest [external]
  - ClpPrimalQuadraticDantzig [external]
- ClpSimplexProgress [external]
- ClpSolve [external]
- ClpTrustedData [external]
- CoinAbcAnyFactorization [external]
  - CoinAbcDenseFactorization [external]
  - CoinAbcTypeFactorization [external]
- CoinAbcStack [external]
- CoinAbcStatistics [external]
- CoinAbsFitEq [external]
- CoinArrayWithLength [external]
  - CoinArbitraryArrayWithLength [external]
  - CoinBigIndexArrayWithLength [external]
  - CoinDoubleArrayWithLength [external]
  - CoinFactorizationDoubleArrayWithLength [external]
  - CoinFactorizationLongDoubleArrayWithLength [external]
  - CoinIntArrayWithLength [external]
  - CoinUnsignedIntArrayWithLength [external]
  - CoinVoidStarArrayWithLength [external]
- CoinBaseModel [external]
  - CoinModel [external]

```

    CoinStructuredModel [external]
CoinBuild [external]
CoinDenseVector< T > [external]
CoinError [external]
    CglLandP::NoBasisError [external]
    CglLandP::SimplexInterfaceError [external]
CoinExternalVectorFirstGreater_2< class, class, class > [external]
CoinExternalVectorFirstGreater_3< class, class, class, class > [external]
CoinExternalVectorFirstLess_2< class, class, class > [external]
CoinExternalVectorFirstLess_3< class, class, class, class > [external]
CoinFactorization [external]
CoinFileIOBase [external]
    CoinFileInput [external]
    CoinFileOutput [external]
CoinFirstAbsGreater_2< class, class > [external]
CoinFirstAbsGreater_3< class, class, class > [external]
CoinFirstAbsLess_2< class, class > [external]
CoinFirstAbsLess_3< class, class, class > [external]
CoinFirstGreater_2< class, class > [external]
CoinFirstGreater_3< class, class, class > [external]
CoinFirstLess_2< class, class > [external]
CoinFirstLess_3< class, class, class > [external]
CoinMpsIO::CoinHashLink [external]
ClpHashValue::CoinHashLink [external]
CoinLpIO::CoinHashLink [external]
CoinIndexedVector [external]
    CoinPartitionedVector [external]
    LAP::TabRow [external]
CoinLpIO [external]
CoinMessageHandler [external]
    MyMessageHandler [external]
CoinMessages [external]
    CglMessage [external]
    ClpMessage [external]
    CoinMessage [external]
    LAP::LandPMessages [external]
    LAP::LapMessages [external]
CoinModelHash [external]
CoinModelHash2 [external]
CoinModelHashLink [external]
CoinModelInfo2 [external]
CoinModelLink [external]
CoinModelLinkedList [external]
CoinModelTriple [external]
CoinMpsCardReader [external]
CoinMpsIO [external]
CoinOneMessage [external]
CoinOtherFactorization [external]
    CoinDenseFactorization [external]
    CoinOslFactorization [external]
    CoinSimpFactorization [external]
CoinPackedMatrix [external]

    BCP_lp_relax . . . . . 68
CoinPackedVectorBase [external]

```



CoinPackedVector [external]	
BCP_col . . . . .	34
BCP_row . . . . .	113
CoinShallowPackedVector [external]	
CoinPair< S, T > [external]	
CoinParam [external]	
CoinPrePostsolveMatrix [external]	
CoinPostsolveMatrix [external]	
CoinPresolveMatrix [external]	
CoinPresolveAction [external]	
do_tighten_action [external]	
doubleton_action [external]	
drop_empty_cols_action [external]	
drop_empty_rows_action [external]	
drop_zero_coefficients_action [external]	
dupcol_action [external]	
duprow3_action [external]	
duprow_action [external]	
forcing_constraint_action [external]	
gubrow_action [external]	
implied_free_action [external]	
isolated_constraint_action [external]	
make_fixed_action [external]	
remove_dual_action [external]	
remove_fixed_action [external]	
slack_doubleton_action [external]	
slack_singleton_action [external]	
subst_constraint_action [external]	
triplet_action [external]	
twoxtwo_action [external]	
useless_constraint_action [external]	
CoinPresolveMonitor [external]	
CoinRational [external]	
CoinRelFltEq [external]	
CoinSearchTreeBase [external]	
CoinSearchTree< class > [external]	
CoinSearchTreeCompareBest [external]	
CoinSearchTreeCompareBreadth [external]	
CoinSearchTreeCompareDepth [external]	
CoinSearchTreeComparePreferred [external]	
CoinSearchTreeManager [external]	
CoinSet [external]	
CoinSosSet [external]	
CoinSnapshot [external]	
CoinThreadRandom [external]	
CoinTimer [external]	
CoinTreeNode [external]	
BCP_tm_node . . . . .	126
CoinTreeSiblings [external]	
CoinTriple< S, T, U > [external]	
CoinWarmStart [external]	
CoinWarmStartBasis [external]	
AbcWarmStart [external]	
CoinWarmStartDual [external]	

```

CoinWarmStartPrimalDual[external]
CoinWarmStartVector< T >[external]
CoinWarmStartVector< double >[external]
CoinWarmStartVector< U >[external]
CoinWarmStartVectorPair< T, U >[external]
CoinWarmStartDiff[external]
CoinWarmStartBasisDiff[external]
CoinWarmStartDualDiff[external]
CoinWarmStartPrimalDualDiff[external]
CoinWarmStartVectorDiff< T >[external]
CoinWarmStartVectorDiff< double >[external]
CoinWarmStartVectorDiff< U >[external]
CoinWarmStartVectorPairDiff< T, U >[external]
CoinYacc[external]
std::complex
std::multiset< K >::const_iterator
std::unordered_set< K >::const_iterator
OsiCuts::const_iterator[external]
std::multimap< K, T >::const_iterator
std::basic_string< Char >::const_iterator
std::string::const_iterator
std::wstring::const_iterator
std::deque< T >::const_iterator
std::unordered_map< K, T >::const_iterator
std::forward_list< T >::const_iterator
std::map< K, T >::const_iterator
std::list< T >::const_iterator
std::unordered_multimap< K, T >::const_iterator
std::set< K >::const_iterator
std::unordered_multiset< K >::const_iterator
std::vector< T >::const_iterator
std::set< K >::const_reverse_iterator
std::unordered_multimap< K, T >::const_reverse_iterator
std::forward_list< T >::const_reverse_iterator
std::map< K, T >::const_reverse_iterator
std::basic_string< Char >::const_reverse_iterator
std::string::const_reverse_iterator
std::wstring::const_reverse_iterator
std::deque< T >::const_reverse_iterator
std::list< T >::const_reverse_iterator
std::vector< T >::const_reverse_iterator
std::unordered_map< K, T >::const_reverse_iterator
std::multimap< K, T >::const_reverse_iterator
std::unordered_set< K >::const_reverse_iterator
std::multiset< K >::const_reverse_iterator
std::unordered_multiset< K >::const_reverse_iterator
cut[external]
cut_list[external]
cutParams[external]
LAP::Cuts[external]
cycle[external]
cycle_list[external]
std::deque< T >
std::deque< StdVectorDouble >

```

```
DGG_constraint_t[external]
DGG_data_t[external]
DGG_list_t[external]
disaggregationAction[external]
dropped_zero[external]
dualColumnResult[external]
edge[external]
EKKHlink[external]
std::error_category
std::error_code
std::error_condition
std::exception
    std::bad_alloc
    std::bad_cast
    std::bad_exception
    std::bad_typeid
    std::ios_base::failure
    std::logic_error
        std::domain_error
        std::invalid_argument
        std::length_error
        std::out_of_range
    std::runtime_error
        std::overflow_error
        std::range_error
        std::underflow_error
FactorPointers[external]
std::forward_list< T >
glp_prob[external]
Idiot[external]
IdiotResult[external]
ilp[external]
Info[external]
info_weak[external]
std::ios_base
    basic_ios< char >
    basic_ios< wchar_t >
    std::basic_ios
        basic_istream< char >
        basic_istream< wchar_t >
        basic_ostream< char >
        basic_ostream< wchar_t >
        std::basic_istream
            basic_ifstream< char >
            basic_ifstream< wchar_t >
            basic_iostream< char >
            basic_iostream< wchar_t >
            basic_istreamstream< char >
            basic_istreamstream< wchar_t >
            std::basic_ifstream
                std::ifstream
                std::wifstream
            std::basic_iostream
                basic_fstream< char >
```

```

    basic_fstream< wchar_t >
    basic_stringstream< char >
    basic_stringstream< wchar_t >
    std::basic_fstream
        std::fstream
        std::wfstream
    std::basic_stringstream
        std::stringstream
        std::wstringstream
    std::basic_istream
        std::istream
        std::wistream
    std::basic_ostream
        basic_iostream< char >
        basic_iostream< wchar_t >
        basic_ofstream< char >
        basic_ofstream< wchar_t >
        basic_ostringstream< char >
        basic_ostringstream< wchar_t >
        std::basic_iostream
        std::basic_ofstream
            std::ofstream
            std::wofstream
        std::basic_ostringstream
            std::ostringstream
            std::wostringstream
        std::ostream
        std::wostream
    std::ios
    std::wios
    std::unordered_map< K, T >::iterator
    OsiCuts::iterator[external]
    std::multiset< K >::iterator
    std::unordered_set< K >::iterator
    std::set< K >::iterator
    std::unordered_multimap< K, T >::iterator
    std::multimap< K, T >::iterator
    std::unordered_multiset< K >::iterator
    std::forward_list< T >::iterator
    std::map< K, T >::iterator
    std::wstring::iterator
    std::list< T >::iterator
    std::deque< T >::iterator
    std::string::iterator
    std::basic_string< Char >::iterator
    std::vector< T >::iterator
    std::list< T >
    log_var[external]
    std::map< K, T >
    std::map< int, BCP_cut_algo * >
    std::map< int, BCP_process * >
    std::map< int, BCP_tm_node * >

```

```

std::map< int, BCP_tm_node_to_send * >
std::map< int, BCP_ts_node_data * >
std::map< int, BCP_var_algo * >
std::map< int, Coin::SmartPtr< BCP_cut > >
std::map< int, Coin::SmartPtr< BCP_var > >
std::map< int, double >
std::map< int, int >
std::multimap< K, T >
std::multiset< K >
std::multiset< double >
Options[external]
OsiAuxInfo[external]
    OsiBabSolver[external]
OsiBranchingInformation[external]
OsiBranchingObject[external]
    OsiTwoWayBranchingObject[external]
        OsiIntegerBranchingObject[external]
            BCP_lp_integer_branching_object . . . . . 56
        OsiLotsizeBranchingObject[external]
        OsiSOSBranchingObject[external]
            BCP_lp_sos_branching_object . . . . . 74
OsiChooseVariable[external]
    OsiChooseStrong[external]
OsiCut[external]
    OsiColCut[external]
    OsiRowCut[external]
        OsiRowCut2[external]
OsiCuts[external]
OsiHotInfo[external]
OsiObject[external]
    OsiObject2[external]
        OsiLotsize[external]
        OsiSimpleInteger[external]
        OsiSOS[external]
OsiPresolve[external]
OsiPseudoCosts[external]
OsiRowCutDebugger[external]
OsiSolverBranch[external]
OsiSolverInterface[external]
    OsiClpSolverInterface[external]
    OsiCpxSolverInterface[external]
    OsiGlpkSolverInterface[external]
    OsiGrbSolverInterface[external]
    OsiMskSolverInterface[external]
    OsiSpxSolverInterface[external]
    OsiVolSolverInterface[external]
    OsiXprSolverInterface[external]
OsiSolverResult[external]
Outfo[external]
ClpSimplexOther::parametricsData[external]
parity_ilp[external]
AbcSimplexPrimal::pivotStruct[external]
pool_cut[external]
pool_cut_list[external]

```

presolvehlink[external]	
std::priority_queue< T >	
std::queue< T >	
Coin::ReferencedObject[external]	
BCP_cut . . . . .	38
BCP_cut_algo . . . . .	43
BCP_cut_core . . . . .	44
BCP_node_change . . . . .	96
BCP_user_data . . . . .	142
BCP_var . . . . .	144
BCP_var_algo . . . . .	150
BCP_var_core . . . . .	151
std::unordered_set< K >::reverse_iterator	
std::unordered_map< K, T >::reverse_iterator	
std::forward_list< T >::reverse_iterator	
std::list< T >::reverse_iterator	
std::unordered_multiset< K >::reverse_iterator	
std::map< K, T >::reverse_iterator	
std::wstring::reverse_iterator	
std::unordered_multimap< K, T >::reverse_iterator	
std::deque< T >::reverse_iterator	
std::set< K >::reverse_iterator	
std::basic_string< Char >::reverse_iterator	
std::multiset< K >::reverse_iterator	
std::vector< T >::reverse_iterator	
std::multimap< K, T >::reverse_iterator	
std::string::reverse_iterator	
scatterStruct[external]	
select_cut[external]	
separation_graph[external]	
std::set< K >	
short_path_node[external]	
std::smart_ptr< T >	
Coin::SmartPtr< T >[external]	
Coin::SmartPtr< BCP_cut >[external]	
Coin::SmartPtr< BCP_node_change >[external]	
Coin::SmartPtr< BCP_user_data >[external]	
Coin::SmartPtr< BCP_var >[external]	
std::stack< T >	
symrec[external]	
std::system_error	
OsiUnitTest::TestOutcome[external]	
OsiUnitTest::TestOutcomes[external]	
std::thread	
std::unique_ptr< T >	
std::unordered_map< K, T >	
std::unordered_multimap< K, T >	
std::unordered_multiset< K >	
std::unordered_set< K >	
USER_initialize . . . . .	185
std::valarray< T >	
LAP::Validator[external]	
std::vector< T >	
std::vector< ColumnSelectionStrategy >	

```
std::vector< double >
std::vector< int >
std::vector< OsiObject * >
std::vector< RowSelectionStrategy >
std::vector< std::string >
VOL_alpha_factor[external]
VOL_dual[external]
VOL_dvector[external]
VOL_indc[external]
VOL_ivector[external]
VOL_parms[external]
VOL_primal[external]
VOL_problem[external]
VOL_swing[external]
VOL_user_hooks[external]
    OsiVolSolverInterface[external]
VOL_vh[external]
std::weak_ptr< T >
K
Par
    BCP_parameter_set< Par > . . . . . 99
S
T
U
```





## Chapter 2

# Class Index

### 2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">BCP_buffer</a>	This class describes the message buffer used for all processes of BCP . . . . .	21
<a href="#">BCP_cg_par</a>	Parameters used in the Cut Generator process . . . . .	27
<a href="#">BCP_cg_prob</a>	This class is the central class of the Cut Generator process . . . . .	27
<a href="#">BCP_cg_user</a>	Base class from which the user can derive a problem specific class to be used in the Cut Generator process . . . . .	31
<a href="#">BCP_col</a>	This class holds a column in a compressed form . . . . .	34
<a href="#">BCP_cut</a>	Abstract base class that defines members common to all types of cuts . . . . .	38
<a href="#">BCP_cut_algo</a>	This is the class from which the user should derive her own algorithmic cuts . . . . .	43
<a href="#">BCP_cut_core</a>	Core cuts are the cuts that always stay in the LP formulation . . . . .	44
<a href="#">BCP_cut_set</a>	This class is just a collection of pointers to cuts with a number of methods to manipulate these cuts and/or select certain entries . . . . .	45
<a href="#">BCP_fatal_error</a>	Currently there isn't any error handling in BCP . . . . .	47
<a href="#">BCP_internal_brobj</a>	This class is the internal representation of a branching object . . . . .	48
<a href="#">BCP_lp_branching_object</a>	This class describes a generic branching object . . . . .	51
<a href="#">BCP_lp_cut_pool</a>	. . . . .	56
<a href="#">BCP_lp_integer_branching_object</a>	This class exist only so that we can extract information from <b>OsilIntegerBranchingObject</b> . . . . .	56
<a href="#">BCP_lp_node</a>	NO OLD DOC . . . . .	57
<a href="#">BCP_lp_par</a>	Parameters used in the LP process . . . . .	58

<a href="#">BCP_lp_parent</a>	
NO OLD DOC . . . . .	65
<a href="#">BCP_lp_prob</a>	
NO OLD DOC . . . . .	65
<a href="#">BCP_lp_relax</a>	
An object of type <a href="#">BCP_lp_relax</a> holds the description of an lp relaxation . . . . .	68
<a href="#">BCP_lp_result</a>	
This class holds the results after solving an LP relaxation . . . . .	72
<a href="#">BCP_lp_sos_branching_object</a>	
This class exist only so that we can extract information from <b>OsIntegerBranchingObject</b> . . . . .	74
<a href="#">BCP_lp_statistics</a>	
NO OLD DOC . . . . .	75
<a href="#">BCP_lp_user</a>	
Base class from which the user can derive a problem specific class to be used in the LP process . . . . .	75
<a href="#">BCP_lp_var_pool</a> . . . . .	91
<a href="#">BCP_lp_waiting_col</a> . . . . .	91
<a href="#">BCP_lp_waiting_row</a> . . . . .	91
<a href="#">BCP_MemPool</a> . . . . .	92
<a href="#">BCP_message_environment</a>	
This is an abstract base class that describes the message passing environment . . . . .	92
<a href="#">BCP_node_change</a> . . . . .	96
<a href="#">BCP_node_storage_in_tm</a>	
NO OLD DOC . . . . .	96
<a href="#">BCP_obj_change</a> . . . . .	97
<a href="#">BCP_obj_set_change</a>	
This class stores data about how an object set (set of vars or set of cuts) changes . . . . .	97
<a href="#">BCP_parameter</a>	
This parameter indeintifies a single parameter entry . . . . .	97
<a href="#">BCP_parameter_set&lt; Par &gt;</a>	
This is the class serves as a holder for a set of parameters . . . . .	99
<a href="#">BCP_presolved_lp_brobj</a>	
A presolved branching object candidate . . . . .	102
<a href="#">BCP_problem_core</a>	
This class describes the core of the MIP problem, the variables/cuts in it as well as the matrix corresponding the core variables and cuts . . . . .	105
<a href="#">BCP_problem_core_change</a>	
This class describes changes in the core of the problem . . . . .	108
<a href="#">BCP_process</a> . . . . .	112
<a href="#">BCP_row</a>	
This class holds a row in a compressed form . . . . .	113
<a href="#">BCP_scheduler</a> . . . . .	116
<a href="#">BCP_single_environment</a> . . . . .	118
<a href="#">BCP_slave_params</a>	
NO OLD DOC . . . . .	121
<a href="#">BCP_solution</a>	
This is the abstract base class for a solution to a Mixed Integer Programming problem . . . . .	122
<a href="#">BCP_solution_generic</a>	
This class holds a MIP feasible primal solution . . . . .	123
<a href="#">BCP_string</a>	
This class is a very simple imeplementation of a constant length string . . . . .	125
<a href="#">BCP_tm_flags</a>	
NO OLD DOC . . . . .	125
<a href="#">BCP_tm_node</a> . . . . .	126
<a href="#">BCP_tm_node_data</a> . . . . .	127

<a href="#">BCP_tm_node_to_send</a>	127
<a href="#">BCP_tm_par</a>	
Parameters used in the Tree Manager process	128
<a href="#">BCP_tm_prob</a>	
NO OLD DOC	132
<a href="#">BCP_tm_stat</a>	135
<a href="#">BCP_tm_user</a>	
Base class from which the user can derive a problem specific class to be used in the TM process	135
<a href="#">BCP_tree</a>	
NO OLD DOC	139
<a href="#">BCP_ts_node_data</a>	
Same as <a href="#">BCP_tm_node_data</a> , just there's no need to use smart pointers in this process	140
<a href="#">BCP_ts_par</a>	140
<a href="#">BCP_ts_prob</a>	140
<a href="#">BCP_ts_user</a>	141
<a href="#">BCP_user_class</a>	142
<a href="#">BCP_user_data</a>	142
<a href="#">BCP_user_pack</a>	143
<a href="#">BCP_var</a>	
Abstract base class that defines members common to all types of variables	144
<a href="#">BCP_var_algo</a>	
This is the class from which the user should derive her own algorithmic variables	150
<a href="#">BCP_var_core</a>	
Core variables are the variables that always stay in the LP formulation	151
<a href="#">BCP_var_set</a>	
This class is just a collection of pointers to variables with a number of methods to manipulate these variables and/or select certain entries	152
<a href="#">BCP_vec&lt; T &gt;</a>	
The class <a href="#">BCP_vec</a> serves the same purpose as the vector class in the standard template library	154
<a href="#">BCP_vec_change&lt; T &gt;</a>	
This class stores a vector explicitly or relatively to another vector	164
<a href="#">BCP_vg_par</a>	168
<a href="#">BCP_vg_prob</a>	
This class is the central class of the Variable Generator process	169
<a href="#">BCP_vg_user</a>	
Base class from which the user can derive a problem specific class to be used in the Cut Generator process	173
<a href="#">BCP_warmstart</a>	
Warmstarting information for the LP solver	175
<a href="#">BCP_warmstart_basis</a>	
This class describes a warmstart information that consists of basis information for structural and artificial variables	177
<a href="#">BCP_warmstart_dual</a>	
This class describes a warmstart information that consists solely of the dual vector	180
<a href="#">BCP_warmstart_primaldual</a>	
This class describes a warmstart information that consists solely of the dual vector	183
<a href="#">USER_initialize</a>	
This class is an abstract base class for the initializer class the user has to provide	185



## Chapter 3

# File Index

### 3.1 File List

Here is a list of all documented files with brief descriptions:

BCP_branch.hpp	??
BCP_buffer.hpp	??
BCP_cg.hpp	??
BCP_cg_param.hpp	??
BCP_cg_user.hpp	??
BCP_cut.hpp	??
BCP_enum.hpp	??
BCP_enum_branch.hpp	??
BCP_enum_process_t.hpp	??
BCP_enum_tm.hpp	??
BCP_error.hpp	??
BCP_functions.hpp	??
BCP_indexed_pricing.hpp	??
BCP_lp.hpp	??
BCP_lp_branch.hpp	??
BCP_lp_functions.hpp	??
BCP_lp_main_loop.hpp	??
BCP_lp_node.hpp	??
BCP_lp_param.hpp	??
BCP_lp_pool.hpp	??
BCP_lp_result.hpp	??
BCP_lp_user.hpp	??
BCP_main_fun.hpp	??
BCP_math.hpp	??
BCP_matrix.hpp	??
BCP_mempool.hpp	??
BCP_message.hpp	??
BCP_message_mpi.hpp	??
BCP_message_pvm.hpp	??
BCP_message_single.hpp	??
BCP_message_tag.hpp	??
BCP_node_change.hpp	??
BCP_obj_change.hpp	??
BCP_os.hpp	??

BCP_parameters.hpp	??
BCP_problem_core.hpp	??
BCP_process.hpp	??
BCP_set_intersects.hpp	??
BCP_solution.hpp	??
BCP_string.hpp	??
BCP_tm.hpp	??
BCP_tm_functions.hpp	??
BCP_tm_node.hpp	??
BCP_tm_param.hpp	??
BCP_tm_user.hpp	??
BCP_tmstorage.hpp	??
BCP_USER.hpp	??
BCP_var.hpp	??
BCP_vector.hpp	??
BCP_vector_bool.hpp	??
BCP_vector_change.hpp	??
BCP_vector_char.hpp	??
BCP_vector_double.hpp	??
BCP_vector_general.hpp	??
BCP_vector_int.hpp	??
BCP_vector_sanity.hpp	??
BCP_vector_short.hpp	??
BCP_vg.hpp	??
BCP_vg_param.hpp	??
BCP_vg_user.hpp	??
BCP_warmstart.hpp	??
BCP_warmstart_basis.hpp	??
BCP_warmstart_dual.hpp	??
BCP_warmstart_primaldual.hpp	??
BcpConfig.h	??
config_bcp_default.h	??
config_default.h	??

## Chapter 4

# Class Documentation

### 4.1 BCP\_buffer Class Reference

This class describes the message buffer used for all processes of BCP.

```
#include <BCP_buffer.hpp>
```

#### Public Member Functions

##### Query methods

- BCP\_message\_tag [msgtag](#) () const  
*Return the message tag of the message in the buffer.*
- int [sender](#) () const  
*Return a const pointer to the process id of the sender of the message in the buffer.*
- int [size](#) () const  
*Return the size of the current message in the buffer.*
- const char \* [data](#) () const  
*Return a const pointer to the data stored in the buffer.*

##### Modifying methods

- void [set\\_position](#) (const int pos) throw (BCP\_fatal\_error)  
*Position the read head in the buffer.*
- void [set\\_size](#) (const int s) throw (BCP\_fatal\_error)  
*Cut off the end of the buffer.*
- void [set\\_msgtag](#) (const BCP\_message\_tag tag)  
*Set the message tag on the buffer.*
- void [set\\_content](#) (const char \*data, const size\_t size, int sender, BCP\_message\_tag msgtag)  
*Set the buffer to be a copy of the given data.*
- BCP\_buffer & [operator=](#) (const BCP\_buffer &buf)  
*Make an exact replica of the other buffer.*
- void [make\\_fit](#) (const int add\_size)  
*Reallocate the buffer if necessary so that at least add\_size number of additional bytes will fit into the buffer.*
- void [clear](#) ()  
*Completely clear the buffer.*
- template<class T >  
[BCP\\_buffer & pack](#) (const T &value)

- Pack a single object of type T.*
  - `template<class T >`  
`BCP_buffer & unpack (T &value)`  
*Unpack a single object of type T.*
- `template<class T >`  
`BCP_buffer & pack (const T *const values, const int length)`  
*Pack a C style array of objects of type T.*
- `template<class T >`  
`BCP_buffer & unpack (T *&values, int &length, bool allocate=true) throw (BCP_fatal_error)`  
*Unpack an array of objects of type T, where T **must** be a built-in type (ar at least something that can be copied with memcpy).*
- `BCP_buffer & pack (const BCP_string &value)`  
*Pack a BCP\_string into the buffer.*
  - `BCP_buffer & pack (BCP_string &value)`  
*Pack a BCP\_string into the buffer.*
  - `BCP_buffer & unpack (BCP_string &value)`  
*Unpack a BCP\_string from the buffer.*
- `template<class T >`  
`BCP_buffer & pack (const BCP_vec< T > &vec)`  
*Pack a BCP\_vec into the buffer.*
- `template<class T >`  
`BCP_buffer & pack (const std::vector< T > &vec)`  
*Pack a std::vector into the buffer.*
- `template<class T >`  
`BCP_buffer & unpack (BCP_vec< T > &vec)`  
*Unpack a BCP\_vec from the buffer.*
- `template<class T >`  
`BCP_buffer & unpack (std::vector< T > &vec)`  
*Unpack a std::vector from the buffer.*

## Constructors and destructor

- `BCP_buffer ()`  
*The default constructor creates a buffer of size 16 Kbytes with no message in it.*
- `BCP_buffer (const BCP_buffer &buf)`  
*The copy constructor makes an exact replica of the other buffer.*
- `~BCP_buffer ()`  
*The desctructor deletes all data members (including freeing the buffer).*

## Public Attributes

### Data members

*The data members are public for efficiency reasons.*

*Access these fields sparingly.*

*THINK: maybe it's not that inefficient to access the fields thru functions...*

*Note that:*

1. *The max size of the buffer never decreases. The buffer max size increases when the current max size is not sufficient for the message.*
2. *With normal usage for incoming messages `_size` stays constant while reading out the message and `_pos` moves forward in the buffer.*
3. *With normal usage for outgoing messages `_size` and `_pos` moves forward together as the buffer is filled with the message.*



- BCP\_message\_tag [\\_msgtag](#)  
*The message tag of the last received message.*
- int [\\_sender](#)  
*The process id of the sender of the last received message.*
- size\_t [\\_pos](#)  
*The next read position in the buffer.*
- size\_t [\\_max\\_size](#)  
*The amount of memory allocated for the buffer.*
- size\_t [\\_size](#)  
*The current size of the message (the first `_size` bytes of the buffer).*
- char \* [\\_data](#)  
*Pointer to the buffer itself.*

#### 4.1.1 Detailed Description

This class describes the message buffer used for all processes of BCP.

This buffer is a character array; the components of a message are simply copied into this array one after the other. Note that each process has only one buffer, which serves both for outgoing and incoming messages. This can be done since when a message arrives it is completely unpacked before anything else is done and conversely, once a message is started to be packed together it will be sent out before another message is received.

NOTE: Only the following type of objects can be packed with the various [pack\(\)](#) member methods:

- Objects that can be assigned with `memcpy()`, i.e., built-in non-pointer types, and structs recursively built from such types;
- [BCP\\_vecs](#) of the above and
- [BCP\\_strings](#).

Everything else that needs to be packed at any time must have a [pack\(\)](#) member method.

Definition at line 39 of file `BCP_buffer.hpp`.

#### 4.1.2 Constructor & Destructor Documentation

##### 4.1.2.1 BCP\_buffer::BCP\_buffer ( ) [inline]

The default constructor creates a buffer of size 16 Kbytes with no message in it.

Definition at line 381 of file `BCP_buffer.hpp`.

##### 4.1.2.2 BCP\_buffer::BCP\_buffer ( const BCP\_buffer & buf ) [inline]

The copy constructor makes an exact replica of the other buffer.

Definition at line 385 of file `BCP_buffer.hpp`.

##### 4.1.2.3 BCP\_buffer::~BCP\_buffer ( ) [inline]

The desctructor deletes all data members (including freeing the buffer).

Definition at line 392 of file `BCP_buffer.hpp`.

### 4.1.3 Member Function Documentation

#### 4.1.3.1 `BCP_message_tag BCP_buffer::msgtag ( ) const` `[inline]`

Return the message tag of the message in the buffer.

Definition at line 90 of file `BCP_buffer.hpp`.

#### 4.1.3.2 `int BCP_buffer::sender ( ) const` `[inline]`

Return a const pointer to the process id of the sender of the message in the buffer.

Definition at line 93 of file `BCP_buffer.hpp`.

#### 4.1.3.3 `int BCP_buffer::size ( ) const` `[inline]`

Return the size of the current message in the buffer.

Definition at line 95 of file `BCP_buffer.hpp`.

#### 4.1.3.4 `const char* BCP_buffer::data ( ) const` `[inline]`

Return a const pointer to the data stored in the buffer.

Definition at line 97 of file `BCP_buffer.hpp`.

#### 4.1.3.5 `void BCP_buffer::set_position ( const int pos ) throw BCP_fatal_error` `[inline]`

Position the read head in the buffer.

Must be between 0 and `size()`.

Definition at line 104 of file `BCP_buffer.hpp`.

#### 4.1.3.6 `void BCP_buffer::set_size ( const int s ) throw BCP_fatal_error` `[inline]`

Cut off the end of the buffer.

Must be between 0 and `size()`.

Definition at line 110 of file `BCP_buffer.hpp`.

#### 4.1.3.7 `void BCP_buffer::set_content ( const char * data, const size_t size, int sender, BCP_message_tag msgtag )` `[inline]`

Set the buffer to be a copy of the given data.

Use this with care!

Definition at line 119 of file `BCP_buffer.hpp`.

#### 4.1.3.8 `BCP_buffer& BCP_buffer::operator= ( const BCP_buffer & buf )` `[inline]`

Make an exact replica of the other buffer.

Definition at line 136 of file `BCP_buffer.hpp`.

#### 4.1.3.9 void BCP\_buffer::make\_fit( const int *add\_size* ) [inline]

Reallocate the buffer if necessary so that at least *add\_size* number of additional bytes will fit into the buffer.

Definition at line 153 of file BCP\_buffer.hpp.

#### 4.1.3.10 void BCP\_buffer::clear( ) [inline]

Completely clear the buffer.

Delete and zero out *\_msgtag*, *\_size*, *\_pos* and *\_sender*.

Definition at line 168 of file BCP\_buffer.hpp.

#### 4.1.3.11 template<class T> BCP\_buffer& BCP\_buffer::pack( const T & *value* ) [inline]

Pack a single object of type T.

Copies *sizeof*(T) bytes from the address of the object.

Definition at line 177 of file BCP\_buffer.hpp.

#### 4.1.3.12 template<class T> BCP\_buffer& BCP\_buffer::unpack( T & *value* ) [inline]

Unpack a single object of type T.

Copies *sizeof*(T) bytes to the address of the object.

Definition at line 186 of file BCP\_buffer.hpp.

#### 4.1.3.13 template<class T> BCP\_buffer& BCP\_buffer::pack( const T \*const *values*, const int *length* ) [inline]

Pack a C style array of objects of type T.

Definition at line 197 of file BCP\_buffer.hpp.

#### 4.1.3.14 template<class T> BCP\_buffer& BCP\_buffer::unpack( T \*& *values*, int & *length*, bool *allocate* = true ) throw BCP\_fatal\_error [inline]

Unpack an array of objects of type T, where T **must** be a built-in type (or at least something that can be copied with *memcpy*).

If the third argument is true then memory is allocated for the array and the array pointer and the length of the array are returned in the arguments.

If the third argument is false then the arriving array's length is compared to *length* and an exception is thrown if they are not the same. Also, the array passed as the first argument will be filled with the arriving array.

Definition at line 222 of file BCP\_buffer.hpp.

#### 4.1.3.15 BCP\_buffer& BCP\_buffer::pack( const BCP\_string & *value* ) [inline]

Pack a [BCP\\_string](#) into the buffer.

Definition at line 268 of file BCP\_buffer.hpp.

#### 4.1.3.16 `BCP_buffer& BCP_buffer::pack ( BCP_string & value ) [inline]`

Pack a `BCP_string` into the buffer.

Definition at line 281 of file `BCP_buffer.hpp`.

#### 4.1.3.17 `BCP_buffer& BCP_buffer::unpack ( BCP_string & value ) [inline]`

Unpack a `BCP_string` from the buffer.

Definition at line 294 of file `BCP_buffer.hpp`.

#### 4.1.3.18 `template<class T> BCP_buffer& BCP_buffer::pack ( const BCP_vec<T> & vec ) [inline]`

Pack a `BCP_vec` into the buffer.

Definition at line 304 of file `BCP_buffer.hpp`.

#### 4.1.3.19 `template<class T> BCP_buffer& BCP_buffer::pack ( const std::vector< T> & vec ) [inline]`

Pack a `std::vector` into the buffer.

Definition at line 318 of file `BCP_buffer.hpp`.

#### 4.1.3.20 `template<class T> BCP_buffer& BCP_buffer::unpack ( BCP_vec< T> & vec ) [inline]`

Unpack a `BCP_vec` from the buffer.

Definition at line 332 of file `BCP_buffer.hpp`.

#### 4.1.3.21 `template<class T> BCP_buffer& BCP_buffer::unpack ( std::vector< T> & vec ) [inline]`

Unpack a `std::vector` from the buffer.

Definition at line 354 of file `BCP_buffer.hpp`.

### 4.1.4 Member Data Documentation

#### 4.1.4.1 `BCP_message_tag BCP_buffer::_msgtag`

The message tag of the last *received* message.

This member has no meaning if the buffer holds an outgoing message.

Definition at line 70 of file `BCP_buffer.hpp`.

#### 4.1.4.2 `int BCP_buffer::_sender`

The process id of the sender of the last *received* message.

This member has no meaning if the buffer holds an outgoing message.

Definition at line 73 of file `BCP_buffer.hpp`.

#### 4.1.4.3 `size_t BCP_buffer::_pos`

The next read position in the buffer.

Definition at line 75 of file BCP\_buffer.hpp.

#### 4.1.4.4 `size_t BCP_buffer::_max_size`

The amount of memory allocated for the buffer.

Definition at line 77 of file BCP\_buffer.hpp.

#### 4.1.4.5 `size_t BCP_buffer::_size`

The current size of the message (the first `_size` bytes of the buffer).

Definition at line 80 of file BCP\_buffer.hpp.

#### 4.1.4.6 `char* BCP_buffer::_data`

Pointer to the buffer itself.

Definition at line 82 of file BCP\_buffer.hpp.

The documentation for this class was generated from the following file:

- BCP\_buffer.hpp

## 4.2 BCP\_cg\_par Struct Reference

Parameters used in the Cut Generator process.

```
#include <BCP_cg_param.hpp>
```

Inheritance diagram for BCP\_cg\_par:

## 4.3 BCP\_cg\_prob Class Reference

This class is the central class of the Cut Generator process.

```
#include <BCP_cg.hpp>
```

Inheritance diagram for BCP\_cg\_prob:

Collaboration diagram for BCP\_cg\_prob:

### Public Member Functions

- `BCP_var * unpack_var ()`  
*Unpack a variable.*

### Constructor and destructor

- [BCP\\_cg\\_prob](#) (int my\_id, int parent)  
*The default constructor.*
- virtual [~BCP\\_cg\\_prob](#) ()  
*The destructor deletes everything.*

### Query methods

- bool [has\\_ub](#) () const  
*Return true/false indicating whether any upper bound has been found.*
- double [ub](#) () const  
*Return the current upper bound (BCP\_DBL\_MAX if there's no upper bound found yet.)*

### Modifying methods

- void [ub](#) (const double bd)  
*Set the upper bound equal to the argument.*
- bool [probe\\_messages](#) ()  
*Test if there is a message in the message queue waiting to be processed.*

## Public Attributes

### Data members

- [BCP\\_cg\\_user](#) \* [user](#)  
*The user object holding the user's data.*
- [BCP\\_user\\_pack](#) \* [packer](#)  
*A class that holds the methods about how to pack things.*
- [BCP\\_message\\_environment](#) \* [msg\\_env](#)  
*The message passing environment.*
- [BCP\\_buffer](#) [msg\\_buf](#)  
*The message buffer of the Cut Generator process.*
- [BCP\\_parameter\\_set](#)< [BCP\\_cg\\_par](#) > [par](#)  
*The parameters controlling the Cut Generator process.*
- [BCP\\_problem\\_core](#) \* [core](#)  
*The description of the core of the problem.*
- double [upper\\_bound](#)  
*The proc id of the tree manager.*
- [BCP\\_vec](#)< [BCP\\_var](#) \* > [vars](#)  
*Cuts are to be generated for the LP solution given by these variables and their values (next member).*
- [BCP\\_vec](#)< double > [x](#)  
*The primal values corresponding to the variables above.*
- int [sender](#)  
*The process id of the LP process that sent the solution.*
- int [phase](#)  
*The phase the algorithm is in.*
- int [node\\_level](#)  
*The level of search tree node where the solution was generated.*
- int [node\\_index](#)  
*The index of search tree node where the solution was generated.*
- int [node\\_iteration](#)  
*The iteration within the search tree node where the solution was generated.*

### 4.3.1 Detailed Description

This class is the central class of the Cut Generator process.

Only one object of this type is created and that holds all the data in the CG process. A reference to that object is passed to (almost) every function (or member method) that's invoked within the CG process.

Definition at line 32 of file BCP\_cg.hpp.

### 4.3.2 Constructor & Destructor Documentation

#### 4.3.2.1 BCP\_cg\_prob::BCP\_cg\_prob ( int *my\_id*, int *parent* )

The default constructor.

Initializes every data member to a natural state.

#### 4.3.2.2 virtual BCP\_cg\_prob::~BCP\_cg\_prob ( ) [virtual]

The destructor deletes everything.

### 4.3.3 Member Function Documentation

#### 4.3.3.1 bool BCP\_cg\_prob::has\_ub ( ) const [inline]

Return true/false indicating whether any upper bound has been found.

Definition at line 115 of file BCP\_cg.hpp.

#### 4.3.3.2 void BCP\_cg\_prob::ub ( const double *bd* ) [inline]

Set the upper bound equal to the argument.

Definition at line 124 of file BCP\_cg.hpp.

#### 4.3.3.3 bool BCP\_cg\_prob::probe\_messages ( )

Test if there is a message in the message queue waiting to be processed.

#### 4.3.3.4 BCP\_var\* BCP\_cg\_prob::unpack\_var ( )

Unpack a variable.

Invoked from the built-in [BCP\\_cg\\_user::unpack\\_primal\\_solution\(\)](#).

### 4.3.4 Member Data Documentation

#### 4.3.4.1 BCP\_cg\_user\* BCP\_cg\_prob::user

The user object holding the user's data.

This object is created by a call to the appropriate member of [\[USER\\_initialize\]](#){USER\_initialize.html}.

Definition at line 50 of file BCP\_cg.hpp.

#### 4.3.4.2 **BCP\_user\_pack\*** BCP\_cg\_prob::packer

A class that holds the methods about how to pack things.

Definition at line 53 of file BCP\_cg.hpp.

#### 4.3.4.3 **BCP\_message\_environment\*** BCP\_cg\_prob::msg\_env

The message passing environment.

This object is created by a call to the appropriate member of [\[USER\\_initialize\]](#)([USER\\_initialize.html](#)).

Definition at line 58 of file BCP\_cg.hpp.

#### 4.3.4.4 **BCP\_buffer** BCP\_cg\_prob::msg\_buf

The message buffer of the Cut Generator process.

Definition at line 61 of file BCP\_cg.hpp.

#### 4.3.4.5 **BCP\_parameter\_set**<BCP\_cg\_par> BCP\_cg\_prob::par

The parameters controlling the Cut Generator process.

Definition at line 64 of file BCP\_cg.hpp.

#### 4.3.4.6 **BCP\_problem\_core\*** BCP\_cg\_prob::core

The description of the core of the problem.

Definition at line 67 of file BCP\_cg.hpp.

#### 4.3.4.7 **double** BCP\_cg\_prob::upper\_bound

The proc id of the tree manager.

The best currently known upper bound.

Definition at line 73 of file BCP\_cg.hpp.

#### 4.3.4.8 **BCP\_vec**<BCP\_var\*> BCP\_cg\_prob::vars

Cuts are to be generated for the LP solution given by these variables and their values (next member).

Not all variables need to be listed (e.g., list only those that have fractional values in current LP solution).

See also

[BCP\\_lp\\_user::pack\\_primal\\_solution\(\)](#)

Definition at line 83 of file BCP\_cg.hpp.



4.3.4.9 `BCP_vec<double> BCP_cg_prob::x`

The primal values corresponding to the variables above.

Definition at line 85 of file `BCP_cg.hpp`.

4.3.4.10 `int BCP_cg_prob::sender`

The process id of the LP process that sent the solution.

Definition at line 87 of file `BCP_cg.hpp`.

4.3.4.11 `int BCP_cg_prob::phase`

The phase the algorithm is in.

Definition at line 91 of file `BCP_cg.hpp`.

4.3.4.12 `int BCP_cg_prob::node_level`

The level of search tree node where the solution was generated.

Definition at line 93 of file `BCP_cg.hpp`.

4.3.4.13 `int BCP_cg_prob::node_index`

The index of search tree node where the solution was generated.

Definition at line 95 of file `BCP_cg.hpp`.

4.3.4.14 `int BCP_cg_prob::node_iteration`

The iteration within the search tree node where the solution was generated.

Definition at line 98 of file `BCP_cg.hpp`.

The documentation for this class was generated from the following file:

- `BCP_cg.hpp`

## 4.4 BCP\_cg\_user Class Reference

The `BCP_cg_user` class is the base class from which the user can derive a problem specific class to be used in the Cut Generator process.

```
#include <BCP_cg_user.hpp>
```

Inheritance diagram for `BCP_cg_user`:

Collaboration diagram for `BCP_cg_user`:

## Public Member Functions

- void `send_cut` (const `BCP_cut` &cut)  
*Pack the argument into the message buffer and send it to the sender of the LP solution.*
- virtual void `unpack_module_data` (`BCP_buffer` &buf)  
*Unpack the initial information sent to the Cut Generator process by the Tree Manager.*
- virtual void `unpack_primal_solution` (`BCP_buffer` &buf)  
*Unpack the LP solution arriving from the LP process.*
- virtual void `generate_cuts` (`BCP_vec`< `BCP_var` \* > &vars, `BCP_vec`< double > &x)  
*Perform the actual cut generation.*

## Methods to set and get the pointer to the `BCP_cg_prob`

object.

It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.

- void `setCgProblemPointer` (`BCP_cg_prob` \*ptr)  
*Set the pointer.*
- `BCP_cg_prob` \* `getCgProblemPointer` ()  
*Get the pointer.*

## Informational methods for the user.

- double `upper_bound` () const  
*Return what is the best known upper bound (might be `BCP_DBL_MAX`)*
- int `current_phase` () const  
*Return the phase the algorithm is in.*
- int `current_level` () const  
*Return the level of the search tree node for which cuts are being generated.*
- int `current_index` () const  
*Return the internal index of the search tree node for which cuts are being generated.*
- int `current_iteration` () const  
*Return the iteration count within the search tree node for which cuts are being generated.*

## Methods to get/set BCP parameters on the fly

- char `get_param` (const `BCP_cg_par::chr_params` key) const
- int `get_param` (const `BCP_cg_par::int_params` key) const
- double `get_param` (const `BCP_cg_par::dbl_params` key) const
- const `BCP_string` & `get_param` (const `BCP_cg_par::str_params` key) const
- void `set_param` (const `BCP_cg_par::chr_params` key, const bool val)
- void `set_param` (const `BCP_cg_par::chr_params` key, const char val)
- void `set_param` (const `BCP_cg_par::int_params` key, const int val)
- void `set_param` (const `BCP_cg_par::dbl_params` key, const double val)
- void `set_param` (const `BCP_cg_par::str_params` key, const char \*val)

## Constructor, Destructor

- `BCP_cg_user` ()
- virtual `~BCP_cg_user` ()  
*Being virtual, the destructor invokes the destructor for the real type of the object being deleted.*

### 4.4.1 Detailed Description

The [BCP\\_cg\\_user](#) class is the base class from which the user can derive a problem specific class to be used in the Cut Generator process.

In that derived class the user can store data to be used in the methods she overrides. Also that is the object the user must return in the `USER_initialize::cg_init()` method.

There are two kind of methods in the class. The non-virtual methods are helper functions for the built-in defaults, but the user can use them as well. The virtual methods execute steps in the BCP algorithm where the user might want to override the default behavior.

The default implementations fall into three major categories.

- Empty; doesn't do anything and immediately returns (e.g., [unpack\\_module\\_data\(\)](#)).
- There is no reasonable default, so throw an exception. This happens if the parameter settings drive the flow of in a way that BCP can't perform the necessary function. This behavior is correct since such methods are invoked only if the parameter settings drive the flow of the algorithm that way, in which case the user better implement those methods. (At the momemnt there is no such method in CG.)
- A default is given. Frequently there are multiple defaults and parameters govern which one is selected (e.g., [unpack\\_primal\\_solution\(\)](#)).

Definition at line 50 of file `BCP_cg_user.hpp`.

### 4.4.2 Constructor & Destructor Documentation

4.4.2.1 `virtual BCP_cg_user::~BCP_cg_user( ) [inline], [virtual]`

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 118 of file `BCP_cg_user.hpp`.

### 4.4.3 Member Function Documentation

4.4.3.1 `void BCP_cg_user::send_cut( const BCP_cut & cut )`

Pack the argument into the message buffer and send it to the sender of the LP solution.

Whenever the user generates a cut in the [generate\\_cuts\(\)](#) method she should invoke this method to immediately send off the cut to the LP process.

4.4.3.2 `virtual void BCP_cg_user::unpack_module_data( BCP_buffer & buf ) [virtual]`

Unpack the initial information sent to the Cut Generator process by the Tree Manager.

This information was packed by the method [BCP\\_tm\\_user::pack\\_module\\_data\(\)](#) invoked with `BCP_ProcessType←_CG` as the third (target process type) argument.

Default: empty method.

4.4.3.3 `virtual void BCP_cg_user::unpack_primal_solution( BCP_buffer & buf ) [virtual]`

Unpack the LP solution arriving from the LP process.

This method is invoked only if the user packs the info necessary for cut generation by herself, i.e., she overrides the `BCP_lp_user::pack_primal_solution()`. If that's the case the user has to unpack the same info she has packed in the LP process.

```
4.4.3.4 virtual void BCP_cg_user::generate_cuts ( BCP_vec< BCP_var * > & vars, BCP_vec< double > & x )
        [virtual]
```

Perform the actual cut generation.

Whenever a cut is generated, the user should invoke the `send_cut()` method to send the generated cut back to the LP process.

The documentation for this class was generated from the following file:

- BCP\_cg\_user.hpp

## 4.5 BCP\_col Class Reference

This class holds a column in a compressed form.

```
#include <BCP_matrix.hpp>
```

Inheritance diagram for BCP\_col:

Collaboration diagram for BCP\_col:

### Public Member Functions

#### Query methods

- double `Objective` () const  
*Return the objective coefficient.*
- double `LowerBound` () const  
*Return the lower bound.*
- double `UpperBound` () const  
*Return the upper bound.*

#### General modifying methods

- void `Objective` (const double obj)  
*Set the objective coefficient to the given value.*
- void `LowerBound` (const double lb)  
*Set the lower bound to the given value.*
- void `UpperBound` (const double ub)  
*Set the upper bound to the given value.*
- `BCP_col & operator=` (const `BCP_col` &x)  
*Assignment operator: copy over the contents of x.*
- void `assign` (const int size, int \*&ElementIndices, double \*&ElementValues, const double Obj, const double LB, const double UB)  
*Set the objective coefficient, lower and upper bounds to the given values.*
- void `copy` (const int size, const int \*&ElementIndices, const double \*&ElementValues, const double Obj, const double LB, const double UB)  
*Copy the arguments into the appropriate data members.*

- void `copy` (`BCP_vec`< int >::const\_iterator firstind, `BCP_vec`< int >::const\_iterator lastind, `BCP_vec`< double >::const\_iterator firstval, `BCP_vec`< double >::const\_iterator lastval, const double Obj, const double LB, const double UB)

*Same as the other `copy()` method, except that instead of using vectors the indices (values) are given in `[firstind, lastind)` (`[firstval, lastval)`).*

### Constructors / Destructor

- `BCP_col` ()  
*The default constructor creates an empty column with 0 as objective coefficient, 0.0 as lower and +infinity as upper bound.*
- `BCP_col` (const `BCP_col` &x)  
*The copy constructor makes a copy of x.*
- `BCP_col` (`BCP_vec`< int >::const\_iterator firstind, `BCP_vec`< int >::const\_iterator lastind, `BCP_vec`< double >::const\_iterator firstval, `BCP_vec`< double >::const\_iterator lastval, const double Obj, const double LB, const double UB)  
*This constructor acts exactly like the `copy` method with the same argument list.*
- `BCP_col` (const int size, int \*&ElementIndices, double \*&ElementValues, const double Obj, const double LB, const double UB)  
*This constructor acts exactly like the `assign` method with the same argument list.*
- `BCP_col` (const `CoinPackedVectorBase` &vec, const double Obj, const double LB, const double UB)
- `~BCP_col` ()  
*The destructor deletes all data members.*

### Protected Attributes

#### Data members

- double `_Objective`  
*The objective function coefficient corresponding to the column.*
- double `_LowerBound`  
*The lower bound corresponding to the column.*
- double `_UpperBound`  
*The upper bound corresponding to the column.*

### 4.5.1 Detailed Description

This class holds a column in a compressed form.

That is, it is a packed vector with an objective coefficient, lower and upper bound.

Definition at line 26 of file `BCP_matrix.hpp`.

### 4.5.2 Constructor & Destructor Documentation

#### 4.5.2.1 `BCP_col::BCP_col( )` [inline]

The default constructor creates an empty column with 0 as objective coefficient, 0.0 as lower and +infinity as upper bound.

Definition at line 113 of file `BCP_matrix.hpp`.

#### 4.5.2.2 `BCP_col::BCP_col ( const BCP_col & x ) [inline]`

The copy constructor makes a copy of `x`.

Definition at line 116 of file `BCP_matrix.hpp`.

#### 4.5.2.3 `BCP_col::BCP_col ( BCP_vec< int >::const_iterator firstind, BCP_vec< int >::const_iterator lastind, BCP_vec< double >::const_iterator firstval, BCP_vec< double >::const_iterator lastval, const double Obj, const double LB, const double UB ) [inline]`

This constructor acts exactly like the `copy` method with the same argument list.

Definition at line 121 of file `BCP_matrix.hpp`.

#### 4.5.2.4 `BCP_col::BCP_col ( const int size, int *& ElementIndices, double *& ElementValues, const double Obj, const double LB, const double UB ) [inline]`

This constructor acts exactly like the `assign` method with the same argument list.

Definition at line 131 of file `BCP_matrix.hpp`.

#### 4.5.2.5 `BCP_col::~~BCP_col ( ) [inline]`

The destructor deletes all data members.

Definition at line 143 of file `BCP_matrix.hpp`.

### 4.5.3 Member Function Documentation

#### 4.5.3.1 `double BCP_col::Objective ( ) const [inline]`

Return the objective coefficient.

Definition at line 44 of file `BCP_matrix.hpp`.

#### 4.5.3.2 `double BCP_col::LowerBound ( ) const [inline]`

Return the lower bound.

Definition at line 46 of file `BCP_matrix.hpp`.

#### 4.5.3.3 `double BCP_col::UpperBound ( ) const [inline]`

Return the upper bound.

Definition at line 48 of file `BCP_matrix.hpp`.

#### 4.5.3.4 `void BCP_col::Objective ( const double obj ) [inline]`

Set the objective coefficient to the given value.

Definition at line 55 of file `BCP_matrix.hpp`.

4.5.3.5 `void BCP_col::LowerBound ( const double lb ) [inline]`

Set the lower bound to the given value.

Definition at line 57 of file BCP\_matrix.hpp.

4.5.3.6 `void BCP_col::UpperBound ( const double ub ) [inline]`

Set the upper bound to the given value.

Definition at line 59 of file BCP\_matrix.hpp.

4.5.3.7 `BCP_col& BCP_col::operator= ( const BCP_col & x ) [inline]`

Assignment operator: copy over the contents of *x*.

Definition at line 62 of file BCP\_matrix.hpp.

4.5.3.8 `void BCP_col::assign ( const int size, int * ElementIndices, double * ElementValues, const double Obj, const double LB, const double UB ) [inline]`

Set the objective coefficient, lower and upper bounds to the given values.

Also invokes the assign method of the underlying packed vector.

Definition at line 73 of file BCP\_matrix.hpp.

4.5.3.9 `void BCP_col::copy ( const int size, const int * ElementIndices, const double * ElementValues, const double Obj, const double LB, const double UB ) [inline]`

Copy the arguments into the appropriate data members.

Definition at line 83 of file BCP\_matrix.hpp.

4.5.3.10 `void BCP_col::copy ( BCP_vec< int >::const_iterator firstind, BCP_vec< int >::const_iterator lastind, BCP_vec< double >::const_iterator firstval, BCP_vec< double >::const_iterator lastval, const double Obj, const double LB, const double UB ) [inline]`

Same as the other `copy()` method, except that instead of using vectors the indices (values) are given in [*firstind*, *lastind*) ([*firstval*, *lastval*)).

Definition at line 95 of file BCP\_matrix.hpp.

## 4.5.4 Member Data Documentation

4.5.4.1 `double BCP_col::_Objective [protected]`

The objective function coefficient corresponding to the column.

Definition at line 32 of file BCP\_matrix.hpp.

#### 4.5.4.2 double BCP\_col::\_LowerBound [protected]

The lower bound corresponding to the column.

Definition at line 34 of file BCP\_matrix.hpp.

#### 4.5.4.3 double BCP\_col::\_UpperBound [protected]

The upper bound corresponding to the column.

Definition at line 36 of file BCP\_matrix.hpp.

The documentation for this class was generated from the following file:

- BCP\_matrix.hpp

## 4.6 BCP\_cut Class Reference

Abstract base class that defines members common to all types of cuts.

```
#include <BCP_cut.hpp>
```

Inheritance diagram for BCP\_cut:

Collaboration diagram for BCP\_cut:

### Public Member Functions

#### Constructor and destructor

*Note that there is no default constructor.*

*There is no such thing as "default cut".*

- [BCP\\_cut](#) (const double [lb](#), const double [ub](#))  
*The constructor sets the internal index of the cut to zero and the other data members to the given arguments.*
- virtual [~BCP\\_cut](#) ()  
*The destructor is virtual so that the appropriate destructor is invoked for every cut.*

#### Query methods

- virtual BCP\_object\_t [obj\\_type](#) () const =0  
*Return the type of the variable.*
- int [effective\\_count](#) () const  
*Return the effectiveness count of the cut (only in LP process).*
- double [lb](#) () const  
*Return the lower bound on the cut.*
- double [ub](#) () const  
*Return the upper bound on the cut.*
- int [bcpind](#) () const  
*Return the internal index of the cut.*
- BCP\_obj\_status [status](#) () const  
*Return the status of the cut.*
- bool [dont\\_send\\_to\\_pool](#) () const  
*Return whether the cut should be sent to the Cut Pool process.*
- bool [is\\_non\\_removable](#) () const



*Return whether the cut marked as NotRemovable.*

- bool `is_to_be_removed` () const

*Return whether the cut must be removed from the formulation.*

### Modifying methods

- void `set_effective_count` (const int cnt)  
*Set the effectiveness count to the given value.*
- int `increase_effective_count` ()  
*Increase the effectiveness count by 1 (or to 1 if it was negative).*
- int `decrease_effective_count` ()  
*Decrease the effectiveness count by 1 (or to -1 if it was positive).*
- void `set_lb` (const double lb)  
*Set the lower bound on the cut.*
- void `set_ub` (const double ub)  
*Set the upper bound on the cut.*
- void `change_lb_ub_st` (const BCP\_obj\_change &change)  
*Set the lower/upper bounds and the status of the cut simultaneously to the values given in the data members of the argument.*
- void `change_bounds` (const double lb, const double ub)  
*Change just the lower/upper bounds.*
- void `set_bcpind` (const int bcpind)  
*Set the internal index of the cut.*
- void `set_bcpind_flip` ()  
*Flip the internal index of the variable to its negative.*
- void `set_status` (const BCP\_obj\_status stat)  
*Set the status of the cut.*
- void `dont_send_to_pool` (bool flag)  
*Set/unset the flag controlling whether the cut could be sent to the Cut Pool process.*
- void `make_active` ()  
*Mark the cut as active.*
- void `make_non_removable` ()  
*Mark the cut as NotRemovable.*
- void `make_to_be_removed` ()  
*Mark the cut as ToBeRemoved.*

### Protected Attributes

- double `_lb`  
*Lower bound of the cut.*
- double `_ub`  
*Upper bound of the cut.*

#### 4.6.1 Detailed Description

Abstract base class that defines members common to all types of cuts.

Classes describing the three types of cuts (core and algorithmic) are derived from this class. No object of type `BCP_cut` can exist (having purely virtual members in the class enforces this restriction).

Definition at line 29 of file `BCP_cut.hpp`.

## 4.6.2 Constructor & Destructor Documentation

### 4.6.2.1 `BCP_cut::BCP_cut ( const double lb, const double ub ) [inline]`

The constructor sets the internal index of the cut to zero and the other data members to the given arguments.

Definition at line 68 of file `BCP_cut.hpp`.

### 4.6.2.2 `virtual BCP_cut::~BCP_cut ( ) [inline], [virtual]`

The destructor is virtual so that the appropriate destructor is invoked for every cut.

Definition at line 72 of file `BCP_cut.hpp`.

## 4.6.3 Member Function Documentation

### 4.6.3.1 `virtual BCP_object_t BCP_cut::obj_type ( ) const [pure virtual]`

Return the type of the variable.

Implemented in [BCP\\_cut\\_algo](#), and [BCP\\_cut\\_core](#).

### 4.6.3.2 `int BCP_cut::effective_count ( ) const [inline]`

Return the effectiveness count of the cut (only in LP process).

Definition at line 80 of file `BCP_cut.hpp`.

### 4.6.3.3 `double BCP_cut::lb ( ) const [inline]`

Return the lower bound on the cut.

Definition at line 82 of file `BCP_cut.hpp`.

### 4.6.3.4 `double BCP_cut::ub ( ) const [inline]`

Return the upper bound on the cut.

Definition at line 84 of file `BCP_cut.hpp`.

### 4.6.3.5 `int BCP_cut::bcpind ( ) const [inline]`

Return the internal index of the cut.

Definition at line 86 of file `BCP_cut.hpp`.

### 4.6.3.6 `BCP_obj_status BCP_cut::status ( ) const [inline]`

Return the status of the cut.

Definition at line 91 of file `BCP_cut.hpp`.

#### 4.6.3.7 `bool BCP_cut::dont_send_to_pool( ) const [inline]`

Return whether the cut should be sent to the Cut Pool process.

(Assuming that it stays in the formulation long enough to qualify to be sent to the Cut Pool at all.

Definition at line 95 of file BCP\_cut.hpp.

#### 4.6.3.8 `bool BCP_cut::is_non_removable( ) const [inline]`

Return whether the cut marked as NotRemovable.

Such cuts are, e.g., the branching cuts.

Definition at line 100 of file BCP\_cut.hpp.

#### 4.6.3.9 `bool BCP_cut::is_to_be_removed( ) const [inline]`

Return whether the cut must be removed from the formulation.

There are very few circumstances when this flag is set; all of them are completely internal to BCP.

Definition at line 106 of file BCP\_cut.hpp.

#### 4.6.3.10 `void BCP_cut::set_effective_count( const int cnt ) [inline]`

Set the effectiveness count to the given value.

Definition at line 115 of file BCP\_cut.hpp.

#### 4.6.3.11 `int BCP_cut::increase_effective_count( ) [inline]`

Increase the effectiveness count by 1 (or to 1 if it was negative).

Return the new effectiveness count.

Definition at line 118 of file BCP\_cut.hpp.

#### 4.6.3.12 `int BCP_cut::decrease_effective_count( ) [inline]`

Decrease the effectiveness count by 1 (or to -1 if it was positive).

Return the new effectiveness count.

Definition at line 124 of file BCP\_cut.hpp.

#### 4.6.3.13 `void BCP_cut::set_lb( const double lb ) [inline]`

Set the lower bound on the cut.

Definition at line 129 of file BCP\_cut.hpp.

#### 4.6.3.14 `void BCP_cut::set_ub( const double ub ) [inline]`

Set the upper bound on the cut.

Definition at line 131 of file BCP\_cut.hpp.

**4.6.3.15** void BCP\_cut::change\_lb\_ub\_st ( const BCP\_obj\_change & *change* ) [inline]

Set the lower/upper bounds and the status of the cut simultaneously to the values given in the data members of the argument.

Definition at line 134 of file BCP\_cut.hpp.

**4.6.3.16** void BCP\_cut::change\_bounds ( const double *lb*, const double *ub* ) [inline]

Change just the lower/upper bounds.

Definition at line 142 of file BCP\_cut.hpp.

**4.6.3.17** void BCP\_cut::set\_bcpind ( const int *bcpind* ) [inline]

Set the internal index of the cut.

Definition at line 149 of file BCP\_cut.hpp.

**4.6.3.18** void BCP\_cut::set\_status ( const BCP\_obj\_status *stat* ) [inline]

Set the status of the cut.

Definition at line 156 of file BCP\_cut.hpp.

**4.6.3.19** void BCP\_cut::dont\_send\_to\_pool ( bool *flag* ) [inline]

Set/unset the flag controlling whether the cut could be sent to the Cut Pool process.

Definition at line 159 of file BCP\_cut.hpp.

**4.6.3.20** void BCP\_cut::make\_active ( ) [inline]

Mark the cut as active.

Note that when this method is invoked the lp formulation must be modified as well: the original bounds of the cut must be reset.

Definition at line 167 of file BCP\_cut.hpp.

**4.6.3.21** void BCP\_cut::make\_non\_removable ( ) [inline]

Mark the cut as NotRemovable.

Definition at line 171 of file BCP\_cut.hpp.

**4.6.3.22** void BCP\_cut::make\_to\_be\_removed ( ) [inline]

Mark the cut as ToBeRemoved.

It will actually be removed immediately after all cuts that have to be marked this way are marked.

Definition at line 178 of file BCP\_cut.hpp.

#### 4.6.4 Member Data Documentation

##### 4.6.4.1 double BCP\_cut::\_lb [protected]

Lower bound of the cut.

Definition at line 56 of file BCP\_cut.hpp.

##### 4.6.4.2 double BCP\_cut::\_ub [protected]

Upper bound of the cut.

Definition at line 58 of file BCP\_cut.hpp.

The documentation for this class was generated from the following file:

- BCP\_cut.hpp

## 4.7 BCP\_cut\_algo Class Reference

This is the class from which the user should derive her own algorithmic cuts.

```
#include <BCP_cut.hpp>
```

Inheritance diagram for BCP\_cut\_algo:

Collaboration diagram for BCP\_cut\_algo:

### Public Member Functions

#### Constructor and destructor

- [BCP\\_cut\\_algo](#) (const double [lb](#), const double [ub](#))  
*This constructor just sets the data members to the given values.*
- virtual [~BCP\\_cut\\_algo](#) ()=0  
*The destructor deletes the object.*

#### Query methods

- [BCP\\_object\\_t](#) [obj\\_type](#) () const  
*Return BCP\_AlgoObj indicating that the object is an algorithmic cut.*

### Additional Inherited Members

#### 4.7.1 Detailed Description

This is the class from which the user should derive her own algorithmic cuts.

Note that such an object cannot be constructed (it has pure virtual methods), only objects with types derived from [BCP\\_cut\\_algo](#) can be created. Such objects are constructed either directly by the user or by the unpacking functions of the [BCP\\_xx\\_user](#) classes.

Definition at line 242 of file BCP\_cut.hpp.

## 4.7.2 Constructor & Destructor Documentation

### 4.7.2.1 BCP\_cut\_algo::BCP\_cut\_algo ( const double *lb*, const double *ub* ) [inline]

This constructor just sets the data members to the given values.

See also the constructor of [BCP\\_cut](#).

Definition at line 259 of file BCP\_cut.hpp.

### 4.7.2.2 virtual BCP\_cut\_algo::~BCP\_cut\_algo ( ) [pure virtual]

The destructor deletes the object.

## 4.7.3 Member Function Documentation

### 4.7.3.1 BCP\_object\_t BCP\_cut\_algo::obj\_type ( ) const [inline], [virtual]

Return BCP\_AlgoObj indicating that the object is an algorithmic cut.

Implements [BCP\\_cut](#).

Definition at line 268 of file BCP\_cut.hpp.

The documentation for this class was generated from the following file:

- BCP\_cut.hpp

## 4.8 BCP\_cut\_core Class Reference

Core cuts are the cuts that always stay in the LP formulation.

```
#include <BCP_cut.hpp>
```

Inheritance diagram for BCP\_cut\_core:

Collaboration diagram for BCP\_cut\_core:

## Public Member Functions

### Constructors and desctructor

- [BCP\\_cut\\_core](#) (const [BCP\\_cut\\_core](#) &x)  
*The copy constructor makes a replica of the argument.*
- [BCP\\_cut\\_core](#) (const double *lb*, const double *ub*)  
*This constructor just sets the data members to the given values.*
- [~BCP\\_cut\\_core](#) ()  
*The destructor deletes the object.*

### Query methods

- BCP\_object\_t [obj\\_type](#) () const  
*Return BCP\_CoreObj indicating that the object is a core cut.*

## Additional Inherited Members

### 4.8.1 Detailed Description

Core cuts are the cuts that always stay in the LP formulation.

Therefore the data members in the base class are quite sufficient to describe the cut. The only thing that has to be done here is overriding the pure virtual method [obj\\_type\(\)](#).

Definition at line 195 of file BCP\_cut.hpp.

### 4.8.2 Constructor & Destructor Documentation

#### 4.8.2.1 BCP\_cut\_core::BCP\_cut\_core ( const BCP\_cut\_core & x ) `[inline]`

The copy constructor makes a replica of the argument.

Definition at line 210 of file BCP\_cut.hpp.

#### 4.8.2.2 BCP\_cut\_core::BCP\_cut\_core ( const double lb, const double ub ) `[inline]`

This constructor just sets the data members to the given values.

See also the constructor of [BCP\\_cut](#).

Definition at line 217 of file BCP\_cut.hpp.

#### 4.8.2.3 BCP\_cut\_core::~BCP\_cut\_core ( ) `[inline]`

The destructor deletes the object.

Definition at line 219 of file BCP\_cut.hpp.

### 4.8.3 Member Function Documentation

#### 4.8.3.1 BCP\_object\_t BCP\_cut\_core::obj\_type ( ) const `[inline], [virtual]`

Return BCP\_CoreObj indicating that the object is a core cut.

Implements [BCP\\_cut](#).

Definition at line 226 of file BCP\_cut.hpp.

The documentation for this class was generated from the following file:

- BCP\_cut.hpp

## 4.9 BCP\_cut\_set Class Reference

This class is just a collection of pointers to cuts with a number of methods to manipulate these cuts and/or select certain entries.

```
#include <BCP_cut.hpp>
```

Inheritance diagram for `BCP_cut_set`:

Collaboration diagram for `BCP_cut_set`:

## Public Member Functions

### Constructor and destructor

- `BCP_cut_set()`  
*The default constructor creates a cut set with no cuts in it.*
- `~BCP_cut_set()`  
*The destructor empties the cut set.*

### Modifying methods

- void `append` (const `BCP_vec< BCP_cut * >` &x)  
*Append the cuts in the vector `x` to the end of the cut set.*
- void `append` (`BCP_cut_set::const_iterator` first, `BCP_cut_set::const_iterator` last)  
*Append the cuts in `[first, last)` to the end of the cut set.*
- void `set_lb_ub` (const `BCP_vec< int >` &pos, `BCP_vec< double >::const_iterator` bounds)  
*Set the lower/upper bound pairs of the entries given by the contents of `pos` to the values in `[bounds, bounds + pos)`.*
- void `set_lb_ub_st` (const `BCP_vec< BCP_obj_change >` &cc)  
*Set the lower/upper bound pairs and the stati of the first `cc`.*
- void `set_lb_ub_st` (`BCP_vec< int >::const_iterator` pos, const `BCP_vec< BCP_obj_change >` &cc)  
*Set the lower/upper bound pairs and the stati of the entries given by the content of `[pos, pos + cc)`.*

### Methods related to deleting cuts from the cut set

- void `move_deletable_to_pool` (const `BCP_vec< int >` &deletable\_cuts, `BCP_vec< BCP_cut * >` &pool)  
*Move the cut pointers whose indices are listed in `deletable_cuts` into the pool.*

## Additional Inherited Members

### 4.9.1 Detailed Description

This class is just a collection of pointers to cuts with a number of methods to manipulate these cuts and/or select certain entries.

Definition at line 279 of file `BCP_cut.hpp`.

### 4.9.2 Constructor & Destructor Documentation

#### 4.9.2.1 `BCP_cut_set::BCP_cut_set( )` `[inline]`

The default constructor creates a cut set with no cuts in it.

Definition at line 293 of file `BCP_cut.hpp`.

#### 4.9.2.2 `BCP_cut_set::~BCP_cut_set( )` `[inline]`

The destructor empties the cut set.

*NOTE:* the destructor does NOT delete the cuts the members of the cut set point to.

Definition at line 296 of file `BCP_cut.hpp`.



### 4.9.3 Member Function Documentation

4.9.3.1 void BCP\_cut\_set::append ( const BCP\_vec< BCP\_cut \* > & x ) [inline]

Append the cuts in the vector *x* to the end of the cut set.

Definition at line 304 of file BCP\_cut.hpp.

4.9.3.2 void BCP\_cut\_set::append ( BCP\_cut\_set::const\_iterator first, BCP\_cut\_set::const\_iterator last ) [inline]

Append the cuts in [*first*, *last*) to the end of the cut set.

Definition at line 309 of file BCP\_cut.hpp.

4.9.3.3 void BCP\_cut\_set::set\_lb\_ub ( const BCP\_vec< int > & pos, BCP\_vec< double >::const\_iterator bounds )

Set the lower/upper bound pairs of the entries given by the contents of *pos* to the values in [*bounds*, *bounds* + *pos*.

*size()*).

4.9.3.4 void BCP\_cut\_set::set\_lb\_ub\_st ( const BCP\_vec< BCP\_obj\_change > & cc )

Set the lower/upper bound pairs and the stati of the first *cc*.

*size()* entries to the triplets given in the vector. This method is invoked when the cut set is all the cuts in the current formulation and we want to change the triplets for the core cuts, which are at the beginning of that cut set.

4.9.3.5 void BCP\_cut\_set::set\_lb\_ub\_st ( BCP\_vec< int >::const\_iterator pos, const BCP\_vec< BCP\_obj\_change > & cc )

Set the lower/upper bound pairs and the stati of the entries given by the content of [*pos*, *pos* + *cc*.

*size()*) to the triplets contained in *cc*.

4.9.3.6 void BCP\_cut\_set::move\_deletable\_to\_pool ( const BCP\_vec< int > & deletable\_cuts, BCP\_vec< BCP\_cut \* > & pool )

Move the cut pointers whose indices are listed in *deletable\_cuts* into the *pool*.

Note that this method does NOT compress the cut set, it merely replaces the cut pointers with 0 pointers.

The documentation for this class was generated from the following file:

- BCP\_cut.hpp

## 4.10 BCP\_fatal\_error Class Reference

Currently there isn't any error handling in BCP.

```
#include <BCP_error.hpp>
```

## Public Member Functions

- [BCP\\_fatal\\_error](#) (const std::string &str)  
*The constructor prints out the error message, flushes the stdout buffer and aborts execution.*
- [~BCP\\_fatal\\_error](#) ()  
*The destructor exists only because it must.*

### 4.10.1 Detailed Description

Currently there isn't any error handling in BCP.

When an object of this type is created, the string stored in the character array of the argument is printed out and execution is aborted with `abort()` (thus a core dump is created).

Definition at line 20 of file `BCP_error.hpp`.

### 4.10.2 Constructor & Destructor Documentation

#### 4.10.2.1 `BCP_fatal_error::BCP_fatal_error( const std::string & str ) [inline]`

The constructor prints out the error message, flushes the stdout buffer and aborts execution.

Definition at line 34 of file `BCP_error.hpp`.

#### 4.10.2.2 `BCP_fatal_error::~~BCP_fatal_error( ) [inline]`

The destructor exists only because it must.

Definition at line 50 of file `BCP_error.hpp`.

The documentation for this class was generated from the following file:

- `BCP_error.hpp`

## 4.11 BCP\_internal\_brobj Class Reference

This class is the internal representation of a branching object.

```
#include <BCP_branch.hpp>
```

## Public Member Functions

### Constructors and destructor

- [BCP\\_internal\\_brobj](#) ()  
*The default constructor creates an empty internal branching object (which can be filled later by unpacking a buffer).*
- [BCP\\_internal\\_brobj](#) (BCP\_lp\_branching\_object &candidate)  
*This constructor sets the number of children and copies the contents of the positions and bounds of the forced changes to the positions and bounds of the newly created internal branching object.*
- [~BCP\\_internal\\_brobj](#) ()  
*The desctructor deletes all data members.*

### Query methods

- `int child_num () const`  
*Return the number of children.*
- `int affected_varnum () const`  
*Return the number of affected variables.*
- `int affected_cutnum () const`  
*Return the number of affected cuts.*
- `const BCP_vec< int > & var_positions () const`  
*Return a const reference to the vector of positions of variables affected by the branching.*
- `const BCP_vec< int > & cut_positions () const`  
*Return a const reference to the vector of positions of cuts affected by the branching.*
- `BCP_vec< double >::const_iterator var_bounds_child (const int index) const`  
*Return a const iterator within \_var\_bounds to the location where the bound pairs for the index-th child start.*
- `BCP_vec< double >::const_iterator cut_bounds_child (const int index) const`  
*Return a const iterator within \_cut\_bounds to the location where the bound pairs for the index-th child start.*

### Interaction with the LP solver

- `void apply_child_bounds (OsiSolverInterface *lp, int child_ind) const`  
*Modify the bounds in the LP solver by applying the changes specified for the child\_ind-th child.*

### Packing and unpacking

- `void pack (BCP_buffer &buf) const`  
*Pack the internal branching object into the buffer.*
- `void unpack (BCP_buffer &buf)`  
*Unpack an internal branching object from the buffer.*

#### 4.11.1 Detailed Description

This class is the internal representation of a branching object.

We document it only for the sake of completeness, the user need not worry about it.

An internal branching object is created AFTER all the cuts/variables the branching object wanted to add the relaxation are already added, thus only the bound changes on affected variables are specified.

NOTE: There are only two ways to set up an internal branching object. One is through a constructor that passes on the data members, the second is to set it up with the default constructor then unpack its content from a buffer.

Definition at line 31 of file BCP\_branch.hpp.

#### 4.11.2 Constructor & Destructor Documentation

##### 4.11.2.1 BCP\_internal\_brobj::BCP\_internal\_brobj ( ) [inline]

The default constructor creates an empty internal branching object (which can be filled later by unpacking a buffer).

Definition at line 66 of file BCP\_branch.hpp.

##### 4.11.2.2 BCP\_internal\_brobj::BCP\_internal\_brobj ( BCP\_lp\_branching\_object & candidate )

This constructor sets the number of children and copies the contents of the positions and bounds of the forced changes to the positions and bounds of the newly created internal branching object.

#### 4.11.2.3 `BCP_internal_brobj::~~BCP_internal_brobj ( ) [inline]`

The desctructor deletes all data members.

Definition at line 73 of file `BCP_branch.hpp`.

### 4.11.3 Member Function Documentation

#### 4.11.3.1 `int BCP_internal_brobj::child_num ( ) const [inline]`

Return the number of children.

Definition at line 79 of file `BCP_branch.hpp`.

#### 4.11.3.2 `int BCP_internal_brobj::affected_varnum ( ) const [inline]`

Return the number of affected variables.

Definition at line 81 of file `BCP_branch.hpp`.

#### 4.11.3.3 `int BCP_internal_brobj::affected_cutnum ( ) const [inline]`

Return the number of affected cuts.

Definition at line 83 of file `BCP_branch.hpp`.

#### 4.11.3.4 `const BCP_vec<int>& BCP_internal_brobj::var_positions ( ) const [inline]`

Return a `const` reference to the vector of positions of variables affected by the branching.

Definition at line 87 of file `BCP_branch.hpp`.

#### 4.11.3.5 `const BCP_vec<int>& BCP_internal_brobj::cut_positions ( ) const [inline]`

Return a `const` reference to the vector of positions of cuts affected by the branching.

Definition at line 90 of file `BCP_branch.hpp`.

#### 4.11.3.6 `BCP_vec<double>::const_iterator BCP_internal_brobj::var_bounds_child ( const int index ) const [inline]`

Return a `const` iterator within `_var_bounds` to the location where the bound pairs for the `index`-th child start.

Definition at line 96 of file `BCP_branch.hpp`.

#### 4.11.3.7 `BCP_vec<double>::const_iterator BCP_internal_brobj::cut_bounds_child ( const int index ) const [inline]`

Return a `const` iterator within `_cut_bounds` to the location where the bound pairs for the `index`-th child start.

Definition at line 103 of file `BCP_branch.hpp`.

4.11.3.8 void BCP\_internal\_brobj::apply\_child\_bounds ( OsiSolverInterface \* lp, int child\_ind ) const

Modify the bounds in the LP solver by applying the changes specified for the child\_ind-th child.

4.11.3.9 void BCP\_internal\_brobj::pack ( BCP\_buffer & buf ) const

Pack the internal branching object into the buffer.

4.11.3.10 void BCP\_internal\_brobj::unpack ( BCP\_buffer & buf )

Unpack an internal branching object from the buffer.

The documentation for this class was generated from the following file:

- BCP\_branch.hpp

## 4.12 BCP\_lp\_branching\_object Class Reference

This class describes a generic branching object.

```
#include <BCP_lp_branch.hpp>
```

Collaboration diagram for BCP\_lp\_branching\_object:

### Public Member Functions

#### Constructor and destructor

- [BCP\\_lp\\_branching\\_object](#) (const int children, [BCP\\_vec< BCP\\_var \\* >](#) \* const new\_vars, [BCP\\_vec< BCP\\_cut \\* >](#) \* const new\_cuts, const [BCP\\_vec< int >](#) \* const fvp, const [BCP\\_vec< int >](#) \* const fcp, const [BCP\\_vec< double >](#) \* const fvb, const [BCP\\_vec< double >](#) \* const fcb, const [BCP\\_vec< int >](#) \* const ivp, const [BCP\\_vec< int >](#) \* const icp, const [BCP\\_vec< double >](#) \* const ivb, const [BCP\\_vec< double >](#) \* const icb)

*The constructor makes a copy of each vector passed to it.*

- [BCP\\_lp\\_branching\\_object](#) (const [BCP\\_lp\\_integer\\_branching\\_object](#) &o, const int \*order)
- [BCP\\_lp\\_branching\\_object](#) (const [OsiSolverInterface](#) \*osi, const [BCP\\_lp\\_sos\\_branching\\_object](#) &o, const int \*order)
- void **set\_presolve\_result** (const [BCP\\_vec< double >](#) &objval, const [BCP\\_vec< int >](#) &termcode)
- [~BCP\\_lp\\_branching\\_object](#) ()

*The destructor deletes each vector.*

#### Query methods

- int [vars\\_affected](#) () const  
*Return the number of variables whose bounds are affected by the branching.*
- int [cuts\\_affected](#) () const  
*Return the number of cuts whose bounds are affected by the branching.*
- int [vars\\_added](#) () const  
*Return the number of variables added in the branching.*
- int [cuts\\_added](#) () const  
*Return the number of cuts added in the branching.*
- [BCP\\_vec< double >::const\\_iterator](#) [forced\\_var\\_bd\\_child](#) (const int index) const

Return a const iterator to the position in the forced variable bound changes where the new bounds for the *index*-th child start.

- `BCP_vec< double >::const_iterator forced_cut_bd_child` (const int index) const  
Return a const iterator to the position in the forced cut bound changes where the new bounds for the *index*-th child start.
- `BCP_vec< double >::const_iterator implied_var_bd_child` (const int index) const  
Return a const iterator to the position in the implied variable bound changes where the new bounds for the *index*-th child start.
- `BCP_vec< double >::const_iterator implied_cut_bd_child` (const int index) const  
Return a const iterator to the position in the implied cut bound changes where the new bounds for the *index*-th child start.

## Modifying methods

- void `init_pos_for_added` (const int added\_vars\_start, const int added\_cuts\_start)  
This method "cleans up" the positions and bounds.
- void `apply_child_bd` (`OsiSolverInterface` \*lp, const int child\_ind) const  
This method invokes the appropriate methods of *lp* to apply the forced and implied bound changes of the *child\_ind*-th child.
- void `print_branching_info` (const int orig\_varnum, const double \*x, const double \*obj) const  
This method prints out some information about the branching object.

## Public Attributes

### Data members

- int `child_num`  
The number of children for this branching object.
- `BCP_vec< BCP_var * > * vars_to_add`  
Variables to be added to the formulation.
- `BCP_vec< BCP_cut * > * cuts_to_add`  
Cuts to be added to the formulation.

### Data members referring to forced bound changes. "Forced" means

that the branching rule specifies this change.

- `BCP_vec< int > * forced_var_pos`  
Positions of variables whose bounds change ("forcibly", by branching) in the children.
- `BCP_vec< int > * forced_cut_pos`  
Positions of cuts whose bounds change ("forcibly", by branching) in the children.
- `BCP_vec< double > * forced_var_bd`  
Contains the actual bounds for the variables indexed by *forced\_var\_pos*.
- `BCP_vec< double > * forced_cut_bd`  
Contains the actual bounds for the cuts indexed by *forced\_cut\_pos*.

### Data members referring to implied bound changes. "Implied" means

that these changes could be made by applying the forced changes and then doing logical fixing.

Therefore this information is not recorded in the Tree Manager when it stores the branching rule. However, if there are implied changes, it is useful to specify them, because strong branching may be more effective then.

The interpretation of these data members is identical to their forced counterparts.

- `BCP_vec< int > * implied_var_pos`
- `BCP_vec< int > * implied_cut_pos`

- `BCP_vec< double > * implied_var_bd`
- `BCP_vec< double > * implied_cut_bd`

**Data members referring to presolved values. The user may have**

*presolved the candidate.*

*Then she may store the termcodes and objvals of the children here.*

- `BCP_vec< double > * objval_`
- `BCP_vec< int > * termcode_`

### 4.12.1 Detailed Description

This class describes a generic branching object.

The object may contain variables and cuts to be added to the formulation and it contains the bound changes for each child.

Note that it is unlikely that the user would need any of the member functions. She should simply call its constructor to create the object. The rest is internal to BCP.

Definition at line 70 of file `BCP_lp_branch.hpp`.

### 4.12.2 Constructor & Destructor Documentation

**4.12.2.1** `BCP_lp_branching_object::BCP_lp_branching_object ( const int children, BCP_vec< BCP_var * > *const new_vars, BCP_vec< BCP_cut * > *const new_cuts, const BCP_vec< int > *const fvp, const BCP_vec< int > *const fcp, const BCP_vec< double > *const fvb, const BCP_vec< double > *const fcb, const BCP_vec< int > *const ivp, const BCP_vec< int > *const icp, const BCP_vec< double > *const ivb, const BCP_vec< double > *const icb )` `[inline]`, `[explicit]`

The constructor makes a copy of each vector passed to it.

If a 0 pointer is passed for one of the arguments that means that the vector is empty. `new_{vars, cuts}` contains the variables/cuts to be added and `[fi] [vc] [pb]` contains the forced/implied variable/cut positions/bounds.

Definition at line 153 of file `BCP_lp_branch.hpp`.

**4.12.2.2** `BCP_lp_branching_object::~~BCP_lp_branching_object ( )` `[inline]`

The destructor deletes each vector.

Definition at line 218 of file `BCP_lp_branch.hpp`.

### 4.12.3 Member Function Documentation

**4.12.3.1** `int BCP_lp_branching_object::vars_affected ( )` `const` `[inline]`

Return the number of variables whose bounds are affected by the branching.

(Including both forced and implied changes.)

Definition at line 232 of file `BCP_lp_branch.hpp`.

#### 4.12.3.2 `int BCP_lp_branching_object::cuts_affected ( ) const [inline]`

Return the number of cuts whose bounds are affected by the branching.

(Including both forced and implied changes.)

Definition at line 239 of file BCP\_lp\_branch.hpp.

#### 4.12.3.3 `int BCP_lp_branching_object::vars_added ( ) const [inline]`

Return the number of variables added in the branching.

Definition at line 245 of file BCP\_lp\_branch.hpp.

#### 4.12.3.4 `int BCP_lp_branching_object::cuts_added ( ) const [inline]`

Return the number of cuts added in the branching.

Definition at line 249 of file BCP\_lp\_branch.hpp.

#### 4.12.3.5 `BCP_vec<double>::const_iterator BCP_lp_branching_object::forced_var_bd_child ( const int index ) const [inline]`

Return a const iterator to the position in the forced variable bound changes where the new bounds for the *index*-th child start.

This method should be invoked only if the appropriate data member is non-0.

Definition at line 257 of file BCP\_lp\_branch.hpp.

#### 4.12.3.6 `BCP_vec<double>::const_iterator BCP_lp_branching_object::forced_cut_bd_child ( const int index ) const [inline]`

Return a const iterator to the position in the forced cut bound changes where the new bounds for the *index*-th child start.

This method should be invoked only if the appropriate data member is non-0.

Definition at line 265 of file BCP\_lp\_branch.hpp.

#### 4.12.3.7 `BCP_vec<double>::const_iterator BCP_lp_branching_object::implied_var_bd_child ( const int index ) const [inline]`

Return a const iterator to the position in the implied variable bound changes where the new bounds for the *index*-th child start.

This method should be invoked only if the appropriate data member is non-0.

Definition at line 273 of file BCP\_lp\_branch.hpp.

#### 4.12.3.8 `BCP_vec<double>::const_iterator BCP_lp_branching_object::implied_cut_bd_child ( const int index ) const [inline]`

Return a const iterator to the position in the implied cut bound changes where the new bounds for the *index*-th child start.



This method should be invoked only if the appropriate data member is non-0.

Definition at line 281 of file BCP\_lp\_branch.hpp.

**4.12.3.9** void BCP\_lp\_branching\_object::init\_pos\_for\_added ( const int *added\_vars\_start*, const int *added\_cuts\_start* )

This method "cleans up" the positions and bounds.

First it re-indexes the negative positions starting from *added\_vars\_start* for the variables (and from *added\_cuts\_start* for the cuts). Then it reorders the positions (and follows that ordering with the bounds) so that the positions will be in increasing order.

**4.12.3.10** void BCP\_lp\_branching\_object::apply\_child\_bd ( OsiSolverInterface \* *lp*, const int *child\_ind* ) const

This method invokes the appropriate methods of *lp* to apply the forced and implied bound changes of the *child\_ind*-th child.

**4.12.3.11** void BCP\_lp\_branching\_object::print\_branching\_info ( const int *orig\_varnum*, const double \* *x*, const double \* *obj* ) const

This method prints out some information about the branching object.

(The positions, new bounds, the primal value of the variables.)

#### 4.12.4 Member Data Documentation

**4.12.4.1** int BCP\_lp\_branching\_object::child\_num

The number of children for this branching object.

Definition at line 84 of file BCP\_lp\_branch.hpp.

**4.12.4.2** BCP\_vec<BCP\_var\*>\* BCP\_lp\_branching\_object::vars\_to\_add

Variables to be added to the formulation.

Definition at line 86 of file BCP\_lp\_branch.hpp.

**4.12.4.3** BCP\_vec<BCP\_cut\*>\* BCP\_lp\_branching\_object::cuts\_to\_add

Cuts to be added to the formulation.

Definition at line 88 of file BCP\_lp\_branch.hpp.

**4.12.4.4** BCP\_vec<int>\* BCP\_lp\_branching\_object::forced\_var\_pos

Positions of variables whose bounds change ("forcibly", by branching) in the children.

If a position is non-negative, it refers to a variable in the current LP formulation. If a position is negative (-i), it refers to an added variable (i-1st).

Definition at line 97 of file BCP\_lp\_branch.hpp.

#### 4.12.4.5 `BCP_vec<int>* BCP_lp_branching_object::forced_cut_pos`

Positions of cuts whose bounds change ("forcibly", by branching) in the children.

If a position is non-negative, it refers to a cut in the current LP formulation. If a position is negative (-i), it refers to an added cut (i-1st).

Definition at line 102 of file `BCP_lp_branch.hpp`.

#### 4.12.4.6 `BCP_vec<double>* BCP_lp_branching_object::forced_var_bd`

Contains the actual bounds for the variables indexed by `forced_var_pos`.

List the lower/upper bound pairs for each of the variables, for each child in turn. The length of this vector is thus `child_num * 2 * forced_var_pos.size()`.

Definition at line 107 of file `BCP_lp_branch.hpp`.

#### 4.12.4.7 `BCP_vec<double>* BCP_lp_branching_object::forced_cut_bd`

Contains the actual bounds for the cuts indexed by `forced_cut_pos`.

List the lower/upper bound pairs for each of the cuts, for each child in turn. The length of this vector is thus `child_num * 2 * forced_cut_pos.size()`.

Definition at line 112 of file `BCP_lp_branch.hpp`.

The documentation for this class was generated from the following file:

- `BCP_lp_branch.hpp`

### 4.13 `BCP_lp_cut_pool` Class Reference

Inheritance diagram for `BCP_lp_cut_pool`:

Collaboration diagram for `BCP_lp_cut_pool`:

#### Additional Inherited Members

##### 4.13.1 Detailed Description

Definition at line 47 of file `BCP_lp_pool.hpp`.

The documentation for this class was generated from the following file:

- `BCP_lp_pool.hpp`

### 4.14 `BCP_lp_integer_branching_object` Class Reference

This class exist only so that we can extract information from `OsIntegerBranchingObject`.

```
#include <BCP_lp_branch.hpp>
```

Inheritance diagram for `BCP_lp_integer_branching_object`:

Collaboration diagram for BCP\_lp\_integer\_branching\_object:

#### 4.14.1 Detailed Description

This class exist only so that we can extract information from **OsilIntegerBranchingObject**.

Definition at line 25 of file BCP\_lp\_branch.hpp.

The documentation for this class was generated from the following file:

- BCP\_lp\_branch.hpp

## 4.15 BCP\_lp\_node Class Reference

NO OLD DOC.

```
#include <BCP_lp_node.hpp>
```

Collaboration diagram for BCP\_lp\_node:

### Public Member Functions

#### Constructor and destructor

- **BCP\_lp\_node** ()
- **~BCP\_lp\_node** ()

#### Query methods

- **size\_t varnum** () const
- **size\_t cutnum** () const

#### Modifying methods

- **void clean** ()

### Public Attributes

- **BCP\_user\_data \* user\_data**  
*Data the user wants to pass along with the search tree node.*

#### Data members

- **BCP\_node\_storage\_in\_tm tm\_storage**

#### Process id's ??

- **int cg**
- **int cp**
- **int vg**
- **int vp**

### 4.15.1 Detailed Description

NO OLD DOC.

This class holds the description of the current node itself.

Definition at line 92 of file BCP\_lp\_node.hpp.

### 4.15.2 Member Data Documentation

#### 4.15.2.1 BCP\_user\_data\* BCP\_lp\_node::user\_data

Data the user wants to pass along with the search tree node.

For now it cannot be stored wrt. the parent.

Definition at line 147 of file BCP\_lp\_node.hpp.

The documentation for this class was generated from the following file:

- BCP\_lp\_node.hpp

## 4.16 BCP\_lp\_par Struct Reference

Parameters used in the LP process.

```
#include <BCP_lp_param.hpp>
```

Inheritance diagram for BCP\_lp\_par:

### Public Types

- enum `chr_params` {  
[BranchOnCuts](#), [CompareNewCutsToOldOnes](#), [CompareNewVarsToOldOnes](#), [DoReducedCostFixingAtZero](#),  
[DoReducedCostFixingAtAnything](#), [MessagePassingIsSerial](#), [ReportWhenDefaultIsExecuted](#), [SendFathomed](#),  
[NodeDesc](#),  
[NoCompressionAtFathom](#), [LpVerb\\_First](#), [LpVerb\\_AddedCutCount](#), [LpVerb\\_AddedVarCount](#),  
[LpVerb\\_ChildrenInfo](#), [LpVerb\\_ColumnGenerationInfo](#), [LpVerb\\_CutsToCutPoolCount](#), [LpVerb\\_VarsToVarPool](#),  
[Count](#),  
[LpVerb\\_FathomInfo](#), [LpVerb\\_IterationCount](#), [LpVerb\\_RelaxedSolution](#), [LpVerb\\_FinalRelaxedSolution](#),  
[LpVerb\\_LpMatrixSize](#), [LpVerb\\_LpSolutionValue](#), [LpVerb\\_MatrixCompression](#), [LpVerb\\_NodeTime](#),  
[LpVerb\\_PresolvePositions](#), [LpVerb\\_PresolveResult](#), [LpVerb\\_ProcessedNodeIndex](#), [LpVerb\\_ReportCutGen](#),  
[Timeout](#),  
[LpVerb\\_ReportVarGenTimeout](#), [LpVerb\\_ReportLocalCutPoolSize](#), [LpVerb\\_ReportLocalVarPoolSize](#), [LpVerb\\_](#),  
[RepricingResult](#),  
[LpVerb\\_VarTightening](#), [LpVerb\\_RowEffectivenessCount](#), [LpVerb\\_StrongBranchPositions](#), [LpVerb\\_Strong](#),  
[BranchResult](#),  
[LpVerb\\_GeneratedCutCount](#), [LpVerb\\_GeneratedVarCount](#), [LpVerb\\_Last](#) }

*Character parameters.*

- enum `int_params` {

NiceLevel, ScaleMatrix, SlackCutDiscardingStrategy, CutEffectiveCountBeforePool, CutPoolCheckFrequency, VarPoolCheckFrequency, IneffectiveConstraints, IneffectiveBeforeDelete, MaxNonDualFeasToAdd\_Min, MaxNonDualFeasToAdd\_Max, CutViolationNorm, MaxCutsAddedPerIteration, MaxVarsAddedPerIteration, MaxLeftoverCutNum, DeletedColToCompress\_Min, DeletedRowToCompress\_Min, MaxPresolverIter, StrongBranchNum, StrongBranch\_CloseToHalfNum, BranchingObjectComparison, ChildPreference, FeasibilityTest, WarmstartInfo, InfoForCG, InfoForVG }

*Integer parameters.*

- enum `dbl_params` { Granularity, DeletedColToCompress\_Frac, DeletedRowToCompress\_Frac, MaxNonDualFeasToAdd\_Frac, MaxLeftoverCutFrac, IntegerTolerance, FirstLP\_FirstCutTimeout, LaterLP\_FirstCutTimeout, FirstLP\_AllCutsTimeout, LaterLP\_AllCutsTimeout, FirstLP\_FirstVarTimeout, LaterLP\_FirstVarTimeout, FirstLP\_AllVarsTimeout, LaterLP\_AllVarsTimeout, MaxRunTime }

*Double parameters.*

- enum `str_params` { LogFileName }

*String parameters.*

- enum `str_array_params`

*There are no string array parameters.*

### Verbosity Parameters.

These are all true/false parameters; if the parameter value is set then the corresponding information is printed.

## 4.16.1 Detailed Description

Parameters used in the LP process.

These parameters can be set in the original parameter file by including the following line:

```
BCP_{parameter name} {parameter value}.
```

Definition at line 14 of file BCP\_Ip\_param.hpp.

## 4.16.2 Member Enumeration Documentation

### 4.16.2.1 enum BCP\_Ip\_par::chr\_params

Character parameters.

All of these variables are used as booleans (true = 1, false = 0).

Enumerator

**BranchOnCuts** If true, BCP supports branching on cuts by providing potential branching candidates for the user. These are cuts that were added to the formulation at some point but became slack later in subsequent LP relaxations.

Values: true (1), false (0). Default: 0.

**CompareNewCutsToOldOnes** If true then the LP process will check each newly received cut whether it already exists in the local cut pool or not.

Values: true (1), false (0). Default: 1.

**CompareNewVarsToOldOnes** If true then the LP process will check each newly arrived variable whether it already exists in the local variable pool or not.

Values: true (1), false (0). Default: 1.

**DoReducedCostFixingAtZero** If true the BCP will attempt to do reduced cost fixing only for variables currently at zero.

Values: true (1), false (0). Default: 1.

**DoReducedCostFixingAtAnything** If true the BCP will attempt to do reduced cost fixing for any variable, no matter what is their current value.

Values: true (1), false (0). Default: 1.

**MessagePassingIsSerial** Indicates whether message passing is serial (all processes are on the same processor) or not.

Values: true (1), false (0). Default: 1.

**ReportWhenDefaultIsExecuted** Print out a message when the default version of an overridable method is executed. Default: 1.

**SendFathomedNodeDesc** Whether to send back the description of fathomed search tree nodes to the Tree Manager. (Might be needed to write proof of optimality.)

Values: true (1), false (0). Default: 1.

**NoCompressionAtFathom** Whether we should refrain from compressing the problem description right before a fathomed node's description is sent back to the tree manager.

Values: true (1), false (0). Default: 0.

**LpVerb\_First** Just a marker for the first LpVerb.

**LpVerb\_AddedCutCount** Print the number of cuts added from the local cut pool in the current iteration. (BCP←\_lp\_main\_loop)

**LpVerb\_AddedVarCount** Print the number of variables added from the local variable pool in the current iteration. (BCP←\_lp\_main\_loop)

**LpVerb\_ChildrenInfo** After a branching object is selected print what happens to the presolved children (e.g., fathomed). (BCP←\_print\_brobj\_stat)

**LpVerb\_ColumnGenerationInfo** Print the number of variables generated before resolving the Lp or fathoming a node. (BCP←\_lp\_fathom)

**LpVerb\_CutsToCutPoolCount** Print the number of cuts sent from the LP to the cut pool. (BCP←\_lp\_send\_cuts\_←to\_cp)

**LpVerb\_VarsToVarPoolCount** \*\*\*\* This parameter is not used anywhere!!! \*\*\*\*

**LpVerb\_FathomInfo** Print information related to fathoming. (BCP←\_lp\_main\_loop, BCP←\_lp\_perform\_fathom, BCP←\_lp\_branch) (BCP←\_lp\_fathom)

**LpVerb\_IterationCount** Print the "Starting iteration x" line. (BCP←\_lp\_main\_loop)

**LpVerb\_RelaxedSolution** Turn on the user hook "display\_lp\_solution". (BCP←\_lp\_main\_loop)

**LpVerb\_FinalRelaxedSolution** Turn on the user hook "display\_lp\_solution" for the last LP relaxation solved at a search tree node. (BCP←\_lp\_main\_loop)

**LpVerb\_LpMatrixSize** \*\*\*\* This parameter is not used anywhere!!! \*\*\*\*

**LpVerb\_LpSolutionValue** Print the size of the problem matrix and the LP solution value after resolving the LP. (BCP←\_lp\_main\_loop)

**LpVerb\_MatrixCompression** Print the number of columns and rows that were deleted during matrix compression. (BCP←\_lp\_delete\_cols\_and\_rows)

**LpVerb\_NodeTime** For each tree node print out how much time was spent on it.

**LpVerb\_PresolvePositions** Print detailed information about all the branching candidates during strong branching. LpVerb\_PresolveResult must be set for this parameter to have an effect. (BCP←\_lp\_perform\_strong\_branching)

**LpVerb\_PresolveResult** Print information on the presolved branching candidates during strong branching. (BCP←\_lp\_perform\_strong\_branching)

**LpVerb\_ProcessedNodeIndex** Print the "Processing NODE x on LEVEL y" line. (BCP←\_lp-main\_loop)

- LpVerb\_ReportCutGenTimeout** Print information if receiving cuts is timed out. (BCP\_lp\_generate\_cuts)
- LpVerb\_ReportVarGenTimeout** Print information if receiving variables is timed out. (BCP\_lp\_generate\_vars)
- LpVerb\_ReportLocalCutPoolSize** Print the current number of cuts in the cut pool. This number is printed several times: before and after generating columns at the current iteration, after removing non-essential cuts, etc. (BCP\_lp\_generate\_cuts)
- LpVerb\_ReportLocalVarPoolSize** Similar as above for variables. (BCP\_lp\_generate\_vars)
- LpVerb\_RepricingResult** \*\*\*\* This parameter is not used anywhere!!! \*\*\*\*
- LpVerb\_VarTightening** Print the number of variables whose bounds have been changed by reduced cost fixing or logical fixing. (BCP\_lp\_fix\_vars)
- LpVerb\_RowEffectivenessCount** Print the number of ineffective rows in the current problem. The definition of what rows are considered ineffective is determined by the parameter IneffectiveConstraints. (BCP\_lp\_adjust\_row\_effectiveness)
- LpVerb\_StrongBranchPositions** Print detailed information on the branching candidate selected by strong branching. LpVerb\_StrongBranchResult must be set for this parameter to have an effect. (BCP\_print\_brobj\_stat)
- LpVerb\_StrongBranchResult** Print information on the branching candidate selected by strong branching. (BCP\_print\_brobj\_stat)
- LpVerb\_GeneratedCutCount** Print the number of cuts generated during this iteration (since the LP was resolved last time). (BCP\_lp\_main\_loop)
- LpVerb\_GeneratedVarCount** Print the number of variables generated during this iteration. (BCP\_lp\_main\_loop)
- LpVerb\_Last** Just a marker for the last LpVerb.

Definition at line 17 of file BCP\_lp\_param.hpp.

#### 4.16.2.2 enum BCP\_lp\_par::int\_params

Integer parameters.

Enumerator

- NiceLevel** What should be the "niceness" of the LP process. In a \*nix environment the LP process will be reniced to this level.  
Values: whatever the operating system accepts. Default: 0.
- ScaleMatrix** Indicates how matrix scaling should be performed. This parameter is directly passed to the LP solver's `load_lp` member function.  
Values: whatever is valid for the LP solver that the user has passed on to BCP. Default: 0.
- SlackCutDiscardingStrategy** The slack cut discarding strategy used in the default version of the function `purge_slack_pool()`.  
Values: BCP\_slack\_cut\_discarding. Default: BCP\_DiscardSlackCutsAtNewIteration
- CutEffectiveCountBeforePool** A cut has to remain effective through this many iterations in the LP before it is sent to the Cut Pool process.  
The default 1000 effectively says that only those cuts are sent to the pool which are effective at branching.  
Values: any positive number. Default: 1000.
- CutPoolCheckFrequency** The Cut Pool is queried for violated valid inequalities after the first LP relaxation is solved and then after every this many iterations.  
Values: any positive number. Default: 10.
- VarPoolCheckFrequency** The Variable Pool is queried for columns that improve the formulation after the first LP relaxation is solved, and then after every this many iterations.  
Values: any positive number. Default: 10.

**IneffectiveConstraints** Indicates which constraints should be considered ineffective.

Values: BCP\_IneffectiveConstraints. Default: BCP\_IneffConstr\_ZeroDualValue.

**IneffectiveBeforeDelete** How many times in a row a constraint must be found ineffective before it is marked for deletion.

Values: any positive number. Default: 1.

**MaxNonDualFeasToAdd\_Min** The number of non dual-feasible columns that can be added at a time to the formulation is a certain fraction (MaxNonDualFeasToAdd\_Frac) of the number of columns in the current formulation. However, if the computed number is outside of the range [MaxNonDualFeasToAdd\_Min, MaxNonDualFeasToAdd\_Max] then it will be set to the appropriate endpoint of the range.

Values: . Default: 5.

**MaxNonDualFeasToAdd\_Max** See the description of the previous parameter.

Values: . Default: 200.

**CutViolationNorm** How cut violation should be computed.

Values: BCP\_CutViolationNorm. Default: BCP\_CutViolationNorm\_Plain (violation in the usual sense).

(The other option is BCP\_CutViolationNorm\_Distance: the distance of the fractional point from the cut.

**MaxCutsAddedPerIteration** The maximum number of violated valid inequalities that can be added per iteration.

Values: . Default: 100,000.

**MaxVarsAddedPerIteration** The maximum number of variables that can be added per iteration.

Values: . Default: 100,000.

**MaxLeftoverCutNum** The maximum number of violated but not added cuts to be kept from one iteration to the next. Also see the MaxLeftoverCutFrac parameter.

**DeletedColToCompress\_Min** The number of columns that must be marked for deletion before matrix compression can occur. Matrix compressions also subject to a minimum number of marked columns as a fraction of the current number of columns (see DeletedColToCompress\_Frac).

Values: positive integer. Default: 10.

**DeletedRowToCompress\_Min** The number of rows that must be marked for deletion before matrix compression can occur. Matrix compression is also subject to a minimum number of marked columns as a fraction of the current number of columns (see DeletedRowToCompress\_Frac).

Values: positive integer. Default: 10.

**MaxPresolverIter** Upper limit on the number of iterations performed in each of the children of the search tree node when presolving branching candidates. This parameter is passed onto the LP solver. If the parameter is set to -1 then the branching candidates are not presolved and the first branching candidate is chosen (if there is any).

Note that for different LP solvers (e.g., simplex based algorithm or the Volume Algorithm) the meaning of an iteration is different, thus this parameter will be set differently.

Values: . Default: 100,000.

**StrongBranchNum** Specifies how many branching variables with values close to half between two integers should be chosen by the built-in branching variable selection routine `select_branching_candidates()` (if this routine is used at all).

Values: . Default: 3.

**StrongBranch\_CloseToHalfNum** **<bf>Deprecated parameter. Use StrongBranchNum</bf>**

**BranchingObjectComparison** Specifies the rule used for built-in branching object comparison (if the built-in routine is used at all).

Values: BCP\_branching\_object\_comparison. Default: BCP\_HighestLowObjval.

**ChildPreference** Specifies the rule used for selecting one of the children of the search tree node for diving.

Values: BCP\_child\_preference. Default: BCP\_PreferChild\_LowBound.

**FeasibilityTest** Specifies which built-in MIP feasibility testing routine should be invoked (if a built-in routine is used at all).

Values: BCP\_feasibility\_test. Default: BCP\_FullTest\_Feasible



**WarmstartInfo** Specifies how warmstart information should be stored in the TM. Possible values are: BCP\_WarmstartNone, BCP\_WarmstartRoot, BCP\_WarmstartParent. The first is obvious, the second means the root node's warmstart info should be sent out along with each node, the last means the parent's warmstart info should be sent out. Default: BCP\_WarmstartParent

**InfoForCG** Indicates what part of the primal solution is sent to the Cut Generator process if the `BCP_lp_user::pack_primal_solution()` method is not overridden.

Values: See BCP\_primal\_solution\_description. Default: BCP\_PrimalSolution\_Nonzeros.

**InfoForVG** Indicates what part of the dual solution is sent to the Variable Generator process if the `BCP_lp_user::pack_dual_solution()` method is not overridden.

Values: See BCP\_dual\_solution\_description. Default: BCP\_DualSolution\_Full.

Definition at line 163 of file BCP\_lp\_param.hpp.

#### 4.16.2.3 enum BCP\_lp\_par::dbl\_params

Double parameters.

##### Enumerator

**Granularity** The minimum difference between the objective value of any two feasible solution (with different objective values).

Values: . Default: .

**DeletedColToCompress\_Frac** The fraction of columns that must be marked for deletion before matrix compression can occur. Matrix compression is also subject to a minimum number of marked columns (see DeletedColToCompress\_Min).

Values: . Default: 1e-8.

**DeletedRowToCompress\_Frac** The fraction of rows that must be marked for deletion before matrix compression can occur. Matrix compression is also subject to a minimum number of marked rows (see DeletedRowToCompress\_Min).

Values: . Default: 1e-8.

**MaxNonDualFeasToAdd\_Frac** The number of non dual-feasible columns that can be added to the formulation at a time cannot exceed this fraction of the the number of columns in the current formulation. However, this limit can be overruled by the hard bounds MaxNonDualFeasToAdd\_Min and MaxNonDualFeasToAdd\_Max (see above).

Default: 0.05.

**MaxLeftoverCutFrac** The maximum fraction of the violated but not added cuts to be kept from one iteration to the next. Also see the MaxLeftoverCutNum parameter.

**IntegerTolerance** Values not further from an integer value than the value of this parameter are considered to be integer.

Values: . Default: .

**FirstLP\_FirstCutTimeout** This and the following three parameters control how long the LP process waits for generated cuts. The parameters specify waiting times (in seconds) separately for the first and later LP relaxations at a search tree node, and also the time to receive the first cut vs all the cuts.

Note that it might make sense to set AllCutsTimeout for a *shorter* time than the first cut time out: "We are willing to wait 5 seconds to receive a cut, but if we do receive a cut the total time we wait is only 2 seconds."

This parameter specifies the time to wait for the first generated cut at the first LP relaxation at a search tree node.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

**LaterLP\_FirstCutTimeout** This parameter specifies the time to wait for the first generated cut at iterations that are not the first at a search tree node. See the remarks at FirstLP\_FirstCutTimeout as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

**FirstLP\_AllCutsTimeout** This parameter specifies the time to wait for cuts at the first LP relaxation at a search tree node. See the remarks at `FirstLP_FirstCutTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

**LaterLP\_AllCutsTimeout** This parameter specifies the time to wait for cuts at iterations that are not the first at a search tree node. See the remarks at `FirstLP_FirstCutTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

**FirstLP\_FirstVarTimeout** This and the following three parameters control how long the LP process waits for generated variables. These parameters are analogous to the ones that control the waiting time for generated cuts. See the remarks at `FirstLP_FirstCutTimeout` as well.

This parameter specifies the time to wait for the first generated variable at the first LP relaxation at a search tree node.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

**LaterLP\_FirstVarTimeout** This parameter specifies the time to wait for the first generated variable at iterations that are not the first at a search tree node. See the remarks at `FirstLP_FirstVarTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

**FirstLP\_AllVarsTimeout** This parameter specifies the time to wait for variables at the first LP relaxation at a search tree node. See the remarks at `FirstLP_FirstVarTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

**LaterLP\_AllVarsTimeout** This parameter specifies the time to wait for variables at iterations that are not the first at a search tree node. See the remarks at `FirstLP_FirstVarTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

**MaxRunTime** Maximum allowed running time.

Definition at line 313 of file `BCP_lp_param.hpp`.

#### 4.16.2.4 enum `BCP_lp_par::str_params`

String parameters.

Enumerator

**LogFileName** The filename all the output should go to.

Definition at line 424 of file `BCP_lp_param.hpp`.

#### 4.16.2.5 enum `BCP_lp_par::str_array_params`

There are no string array parameters.

Definition at line 432 of file `BCP_lp_param.hpp`.

The documentation for this struct was generated from the following file:

- `BCP_lp_param.hpp`

## 4.17 BCP\_lp\_parent Class Reference

NO OLD DOC.

```
#include <BCP_lp_node.hpp>
```

Collaboration diagram for BCP\_lp\_parent:

### Public Member Functions

#### Constructor and destructor

- **BCP\_lp\_parent** ()
- **~BCP\_lp\_parent** ()

#### Modifying methods

- void **clean** ()

### Public Attributes

#### Data members

- [BCP\\_problem\\_core\\_change](#) **core\_as\_change**
- [BCP\\_obj\\_set\\_change](#) **var\_set**  
*this is always explicit, it's just that coding is simpler if we reuse the [BCP\\_obj\\_set\\_change](#) object*
- [BCP\\_obj\\_set\\_change](#) **cut\_set**  
*this is always explicit, it's just that coding is simpler if we reuse the [BCP\\_obj\\_set\\_change](#) object*
- [BCP\\_warmstart](#) \* **warmstart**
- int **index**

### 4.17.1 Detailed Description

NO OLD DOC.

This class holds the description of the parent of the current node.

Definition at line 42 of file BCP\_lp\_node.hpp.

The documentation for this class was generated from the following file:

- BCP\_lp\_node.hpp

## 4.18 BCP\_lp\_prob Class Reference

NO OLD DOC.

```
#include <BCP_lp.hpp>
```

Inheritance diagram for BCP\_lp\_prob:

Collaboration diagram for BCP\_lp\_prob:

## Public Member Functions

### Constructor and destructor

- **BCP\_lp\_prob** (int my\_id, int [parent](#))
- virtual **~BCP\_lp\_prob** ()

### Methods to pack/unpack objects

- void **pack\_var** (const [BCP\\_var](#) &var)
- [BCP\\_var](#) \* **unpack\_var** ()
- void **pack\_cut** (const [BCP\\_cut](#) &cut)
- [BCP\\_cut](#) \* **unpack\_cut** ()

### Accessing parameters

- char **param** ([BCP\\_lp\\_par::chr\\_params](#) key) const
- int **param** ([BCP\\_lp\\_par::int\\_params](#) key) const
- double **param** ([BCP\\_lp\\_par::dbl\\_params](#) key) const
- const [BCP\\_string](#) & **param** ([BCP\\_lp\\_par::str\\_params](#) key) const
- const [BCP\\_vec](#)< [BCP\\_string](#) > & **param** ([BCP\\_lp\\_par::str\\_array\\_params](#) key) const
- double **granularity** () const

### Accessing bounds

- bool **has\_ub** () const
- double **ub** () const
- bool **ub** (double new\_ub)
- bool **over\_ub** (double lb) const

## Public Attributes

- std::vector< **OsiObject** \* > [intAndSosObjects](#)

*Things that can be branched on.*

### User provided members

- [BCP\\_lp\\_user](#) \* **user**
- [BCP\\_user\\_pack](#) \* **packer**
- A class that holds the methods about how to pack things.*
- **OsiSolverInterface** \* **master\_lp**
- **OsiSolverInterface** \* **lp\_solver**
- [BCP\\_message\\_environment](#) \* **msg\_env**

### Parameters

- [BCP\\_parameter\\_set](#)< [BCP\\_lp\\_par](#) > **par**

### Description of the core of the problem

- [BCP\\_problem\\_core](#) \* **core**
- [BCP\\_problem\\_core\\_change](#) \* **core\_as\_change**

### Current search tree node and its parent

- [BCP\\_lp\\_node](#) \* [node](#)  
*Description he current search tree node.*
- [BCP\\_lp\\_parent](#) \* [parent](#)  
*Description of the parent of the current node.*
- [CoinWarmStart](#) \* [warmstartRoot](#)  
*Description of the warmstart info from the end of the root node.*

#### Information needed for processing a node

*Need to be updated when starting a new node.*

- [BCP\\_lp\\_result](#) \* [lp\\_result](#)
- int [var\\_bound\\_changes\\_since\\_logical\\_fixing](#)
- [BCP\\_vec](#)< [BCP\\_cut](#) \* > [slack\\_pool](#)
- [BCP\\_lp\\_var\\_pool](#) \* [local\\_var\\_pool](#)
- [BCP\\_lp\\_cut\\_pool](#) \* [local\\_cut\\_pool](#)
- int [next\\_var\\_index](#)
- int [last\\_var\\_index](#)
- int [next\\_cut\\_index](#)
- int [last\\_cut\\_index](#)

#### Time measurement

- double [start\\_time](#)
- [BCP\\_lp\\_statistics](#) [stat](#)

#### Internal data members

- double [upper\\_bound](#)
- int [phase](#)
- int [no\\_more\\_cuts\\_cnt](#)
- int [no\\_more\\_vars\\_cnt](#)

#### Message passing related fields

- [BCP\\_buffer](#) [msg\\_buf](#)

### 4.18.1 Detailed Description

NO OLD DOC.

Definition at line 102 of file BCP\_lp.hpp.

### 4.18.2 Member Data Documentation

#### 4.18.2.1 [BCP\\_user\\_pack](#)\* [BCP\\_lp\\_prob::packer](#)

A class that holds the methods about how to pack things.

Definition at line 133 of file BCP\_lp.hpp.

#### 4.18.2.2 [std::vector<OsiObject \\*>](#) [BCP\\_lp\\_prob::intAndSosObjects](#)

Things that can be branched on.

If not filled out then BCP scans for them every time a new node is processed.

Definition at line 161 of file BCP\_lp.hpp.

#### 4.18.2.3 `BCP_lp_node*` `BCP_lp_prob::node`

Description he current search tree node.

Definition at line 168 of file `BCP_lp.hpp`.

#### 4.18.2.4 `BCP_lp_parent*` `BCP_lp_prob::parent`

Description of the parent of the current node.

Definition at line 170 of file `BCP_lp.hpp`.

#### 4.18.2.5 `CoinWarmStart*` `BCP_lp_prob::warmstartRoot`

Description of the warmstart info from the end of the root node.

Used only if the `BCP_lp_par::WarmstartInfo` parameter is set to `BCP_WarmstartRoot`.

Definition at line 174 of file `BCP_lp.hpp`.

The documentation for this class was generated from the following file:

- `BCP_lp.hpp`

## 4.19 `BCP_lp_relax` Class Reference

An object of type `BCP_lp_relax` holds the description of an lp relaxation.

```
#include <BCP_matrix.hpp>
```

Inheritance diagram for `BCP_lp_relax`:

Collaboration diagram for `BCP_lp_relax`:

### Public Member Functions

#### Query methods

- `size_t colnum () const`  
*The number of columns.*
- `size_t rownum () const`  
*The number of rows.*
- `const BCP_vec< double > & Objective () const`  
*A const reference to the vector of objective coefficients.*
- `const BCP_vec< double > & ColLowerBound () const`  
*A const reference to the vector of lower bounds on the variables.*
- `const BCP_vec< double > & ColUpperBound () const`  
*A const reference to the vector of upper bounds on the variables.*
- `const BCP_vec< double > & RowLowerBound () const`  
*A const reference to the vector of lower bounds on the cuts.*
- `const BCP_vec< double > & RowUpperBound () const`  
*A const reference to the vector of upper bounds on the cuts.*

#### Methods modifying the whole LP relaxation.

- **BCP\_lp\_relax** & **operator=** (const **BCP\_lp\_relax** &mat)  
*Copy the content of  $x$  into the LP relaxation.*
- void **reserve** (const int MaxColNum, const int MaxRowNum, const int MaxNonzeros)  
*Reserve space in the LP relaxation for at least `MaxColNum` columns, `MaxRowNum` rows and `MaxNonzeros` nonzero entries.*
- void **clear** ()  
*Clear the LP relaxation.*
- void **copyOf** (const **CoinPackedMatrix** &m, const double \*OBJ, const double \*CLB, const double \*CUB, const double \*RLB, const double \*RUB)  
*Set up the LP relaxation by making a copy of the arguments.*
- void **assign** (**CoinPackedMatrix** &m, double \*OBJ, double \*CLB, double \*CUB, double \*RLB, double \*RUB)  
*Set up the LP relaxation by taking over the pointers in the arguments.*

#### Methods for expanding/shrinking the LP relaxation.

- void **erase\_col\_set** (const **BCP\_vec**< int > &pos)  
*Remove the columns whose indices are listed in `pos` from the LP relaxation.*
- void **erase\_row\_set** (const **BCP\_vec**< int > &pos)  
*Remove the rows whose indices are listed in `pos` from the LP relaxation.*

#### Packing/unpacking

- void **pack** (**BCP\_buffer** &buf) const  
*Pack the LP relaxation into the buffer.*
- void **unpack** (**BCP\_buffer** &buf)  
*Unpack the LP relaxation from the buffer.*

#### Constructors and destructor

- **BCP\_lp\_relax** (const bool colordered=true)  
*Create an empty LP relaxation with given ordering.*
- **BCP\_lp\_relax** (const **BCP\_lp\_relax** &mat)  
*The copy constructor makes a copy of the argument LP relaxation.*
- **BCP\_lp\_relax** (**BCP\_vec**< **BCP\_row** \* > &rows, **BCP\_vec**< double > &CLB, **BCP\_vec**< double > &CUB, **BCP\_vec**< double > &OBJ)  
*Create a row major ordered LP relaxation by assigning the content of the arguments to the LP relaxation.*
- **BCP\_lp\_relax** (**BCP\_vec**< **BCP\_row** \* > &rows, **BCP\_vec**< double > &CLB, **BCP\_vec**< double > &CUB, **BCP\_vec**< double > &OBJ, double extra\_gap, double extra\_major)  
*Same as the previous method except that this method allows `extra_gap` and `extra_major` to be specified.*
- **BCP\_lp\_relax** (**BCP\_vec**< **BCP\_col** \* > &cols, **BCP\_vec**< double > &RLB, **BCP\_vec**< double > &RUB)  
*Create a column major ordered LP relaxation by assigning the content of the arguments to the LP relaxation.*
- **BCP\_lp\_relax** (**BCP\_vec**< **BCP\_col** \* > &cols, **BCP\_vec**< double > &RLB, **BCP\_vec**< double > &RUB, double extra\_gap, double extra\_major)  
*Same as the previous method except that this method allows `extra_gap` and `extra_major` to be specified.*
- **BCP\_lp\_relax** (const bool colordered, const **BCP\_vec**< int > &VB, const **BCP\_vec**< int > &EI, const **BCP\_vec**< double > &EV, const **BCP\_vec**< double > &OBJ, const **BCP\_vec**< double > &CLB, const **BCP\_vec**< double > &CUB, const **BCP\_vec**< double > &RLB, const **BCP\_vec**< double > &RUB)  
*Create an LP relaxation of the given ordering by copying the content of the arguments to the LP relaxation.*
- **BCP\_lp\_relax** (const bool colordered, const int rownum, const int colnum, const int nznum, int \*VB, int \*EI, double \*EV, double \*OBJ, double \*CLB, double \*CUB, double \*RLB, double \*RUB)  
*Create an LP relaxation of the given ordering by assigning the content of the arguments to the LP relaxation.*
- **~BCP\_lp\_relax** ()  
*The destructor deletes the data members.*

### 4.19.1 Detailed Description

An object of type `BCP_lp_relax` holds the description of an lp relaxation.

The matrix, lower/upper bounds on the variables and cuts and objective coefficients for the variables.

Definition at line 267 of file `BCP_matrix.hpp`.

### 4.19.2 Constructor & Destructor Documentation

#### 4.19.2.1 `BCP_lp_relax::BCP_lp_relax ( const bool colordered = true ) [inline]`

Create an empty LP relaxation with given ordering.

Definition at line 377 of file `BCP_matrix.hpp`.

#### 4.19.2.2 `BCP_lp_relax::BCP_lp_relax ( const BCP_lp_relax & mat )`

The copy constructor makes a copy of the argument LP relaxation.

#### 4.19.2.3 `BCP_lp_relax::BCP_lp_relax ( BCP_vec< BCP_row * > & rows, BCP_vec< double > & CLB, BCP_vec< double > & CUB, BCP_vec< double > & OBJ )`

Create a row major ordered LP relaxation by assigning the content of the arguments to the LP relaxation.

When the constructor returns the content of 'rows' doesn't change while that of CLB, CUB and OBJ will be whatever the default constructor for those arguments create.

#### 4.19.2.4 `BCP_lp_relax::BCP_lp_relax ( BCP_vec< BCP_row * > & rows, BCP_vec< double > & CLB, BCP_vec< double > & CUB, BCP_vec< double > & OBJ, double extra_gap, double extra_major )`

Same as the previous method except that this method allows `extra_gap` and `extra_major` to be specified.

For the description of those see the documentation of **CoinPackedMatrix**. (`extra_gap` will be used for adding columns, while `extra major` for adding rows)

#### 4.19.2.5 `BCP_lp_relax::BCP_lp_relax ( BCP_vec< BCP_col * > & cols, BCP_vec< double > & RLB, BCP_vec< double > & RUB )`

Create a column major ordered LP relaxation by assigning the content of the arguments to the LP relaxation.

When the constructor returns the content of 'cols' doesn't change while that of RLB and RUB will be whatever the default constructor for those arguments create.

#### 4.19.2.6 `BCP_lp_relax::BCP_lp_relax ( BCP_vec< BCP_col * > & cols, BCP_vec< double > & RLB, BCP_vec< double > & RUB, double extra_gap, double extra_major )`

Same as the previous method except that this method allows `extra_gap` and `extra_major` to be specified.

For the description of those see the documentation of **CoinPackedMatrix**. (`extra_gap` will be used for adding rows, while `extra major` for adding columns)



4.19.2.7 `BCP_lp_relax::BCP_lp_relax ( const bool colordered, const BCP_vec< int > & VB, const BCP_vec< int > & EI, const BCP_vec< double > & EV, const BCP_vec< double > & OBJ, const BCP_vec< double > & CLB, const BCP_vec< double > & CUB, const BCP_vec< double > & RLB, const BCP_vec< double > & RUB )`

Create an LP relaxation of the given ordering by copying the content of the arguments to the LP relaxation.

4.19.2.8 `BCP_lp_relax::BCP_lp_relax ( const bool colordered, const int rownum, const int colnum, const int nznum, int *& VB, int *& EI, double *& EV, double *& OBJ, double *& CLB, double *& CUB, double *& RLB, double *& RUB )`

Create an LP relaxation of the given ordering by assigning the content of the arguments to the LP relaxation.

When the constructor returns the content of the arguments will be NULL pointers.

### 4.19.3 Member Function Documentation

4.19.3.1 `size_t BCP_lp_relax::colnum ( ) const [inline]`

The number of columns.

Definition at line 288 of file BCP\_matrix.hpp.

4.19.3.2 `size_t BCP_lp_relax::rownum ( ) const [inline]`

The number of rows.

Definition at line 290 of file BCP\_matrix.hpp.

4.19.3.3 `const BCP_vec<double>& BCP_lp_relax::Objective ( ) const [inline]`

A const reference to the vector of objective coefficients.

Definition at line 292 of file BCP\_matrix.hpp.

4.19.3.4 `const BCP_vec<double>& BCP_lp_relax::ColLowerBound ( ) const [inline]`

A const reference to the vector of lower bounds on the variables.

Definition at line 294 of file BCP\_matrix.hpp.

4.19.3.5 `const BCP_vec<double>& BCP_lp_relax::ColUpperBound ( ) const [inline]`

A const reference to the vector of upper bounds on the variables.

Definition at line 296 of file BCP\_matrix.hpp.

4.19.3.6 `const BCP_vec<double>& BCP_lp_relax::RowLowerBound ( ) const [inline]`

A const reference to the vector of lower bounds on the cuts.

Definition at line 298 of file BCP\_matrix.hpp.

4.19.3.7 `const BCP_vec<double>& BCP_lp_relax::RowUpperBound ( ) const` `[inline]`

A const reference to the vector of upper bounds on the cuts.

Definition at line 300 of file `BCP_matrix.hpp`.

4.19.3.8 `BCP_lp_relax& BCP_lp_relax::operator= ( const BCP_lp_relax & mat )`

Copy the content of `x` into the LP relaxation.

4.19.3.9 `void BCP_lp_relax::reserve ( const int MaxColNum, const int MaxRowNum, const int MaxNonzeros )`

Reserve space in the LP relaxation for at least `MaxColNum` columns, `MaxRowNum` rows and `MaxNonzeros` nonzero entries.

This is useful when columns/rows are added to the LP relaxation in small chunks and to avoid a series of reallocation we reserve sufficient space up front.

4.19.3.10 `void BCP_lp_relax::clear ( )`

Clear the LP relaxation.

4.19.3.11 `void BCP_lp_relax::erase_col_set ( const BCP_vec< int > & pos )`

Remove the columns whose indices are listed in `pos` from the LP relaxation.

4.19.3.12 `void BCP_lp_relax::erase_row_set ( const BCP_vec< int > & pos )`

Remove the rows whose indices are listed in `pos` from the LP relaxation.

4.19.3.13 `void BCP_lp_relax::pack ( BCP_buffer & buf ) const`

Pack the LP relaxation into the buffer.

4.19.3.14 `void BCP_lp_relax::unpack ( BCP_buffer & buf )`

Unpack the LP relaxation from the buffer.

The documentation for this class was generated from the following file:

- `BCP_matrix.hpp`

## 4.20 BCP\_lp\_result Class Reference

This class holds the results after solving an LP relaxation.

```
#include <BCP_lp_result.hpp>
```

## Public Member Functions

### Constructor and destructor

- [BCP\\_lp\\_result](#) ()  
*The default constructor initializes an empty solution, i.e., one which holds neither an exact nor an approximate solution.*
- [~BCP\\_lp\\_result](#) ()  
*The destructor deletes the data members if they are private copies.*

### Query methods for the solution. These methods (except for

*the first) just return the value of the queried member (in case of the vector members a reference to the vector is returned instead of the pointer.*

- `const std::string & solvname () const`
- `int termcode () const`
- `int iternum () const`
- `double objval () const`
- `const double * x () const`
- `const double * pi () const`
- `const double * dj () const`
- `const double * lhs () const`

### Query methods for general solver information.

- `double primalTolerance () const`  
*Return the primal tolerance of the solver.*
- `double dualTolerance () const`  
*Return the dual tolerance of the solver.*

### Modifying methods

- `void get_results (OsiSolverInterface &lp_solver)`  
*Get the result from the LP solver.*
- `void fake_objective_value (const double val)`  
*Set the lower bound and the exact and approximate objective values to the value given in the argument.*

## 4.20.1 Detailed Description

This class holds the results after solving an LP relaxation.

There may be an exact and/or an approximate solution.

Definition at line 39 of file BCP\_lp\_result.hpp.

## 4.20.2 Constructor & Destructor Documentation

### 4.20.2.1 BCP\_lp\_result::BCP\_lp\_result ( ) [inline]

The default constructor initializes an empty solution, i.e., one which holds neither an exact nor an approximate solution.

Definition at line 87 of file BCP\_lp\_result.hpp.

#### 4.20.2.2 BCP\_lp\_result::~BCP\_lp\_result ( ) [inline]

The destructor deletes the data members if they are private copies.

Definition at line 94 of file BCP\_lp\_result.hpp.

### 4.20.3 Member Function Documentation

#### 4.20.3.1 double BCP\_lp\_result::primalTolerance ( ) const [inline]

Return the primal tolerance of the solver.

Definition at line 135 of file BCP\_lp\_result.hpp.

#### 4.20.3.2 double BCP\_lp\_result::dualTolerance ( ) const [inline]

Return the dual tolerance of the solver.

Definition at line 137 of file BCP\_lp\_result.hpp.

#### 4.20.3.3 void BCP\_lp\_result::get\_results ( OsiSolverInterface & lp\_solver )

Get the result from the LP solver.

Non-vector members will get their values from the LP solver. Vector members are copied out from the LP solver.

#### 4.20.3.4 void BCP\_lp\_result::fake\_objective\_value ( const double val ) [inline]

Set the lower bound and the exact and approximate objective values to the value given in the argument.

Definition at line 148 of file BCP\_lp\_result.hpp.

The documentation for this class was generated from the following file:

- BCP\_lp\_result.hpp

## 4.21 BCP\_lp\_sos\_branching\_object Class Reference

This class exist only so that we can extract information from **OsiIntegerBranchingObject**.

```
#include <BCP_lp_branch.hpp>
```

Inheritance diagram for BCP\_lp\_sos\_branching\_object:

Collaboration diagram for BCP\_lp\_sos\_branching\_object:

### 4.21.1 Detailed Description

This class exist only so that we can extract information from **OsiIntegerBranchingObject**.

Definition at line 38 of file BCP\_lp\_branch.hpp.

The documentation for this class was generated from the following file:

- BCP\_lp\_branch.hpp

## 4.22 BCP\_lp\_statistics Class Reference

NO OLD DOC.

```
#include <BCP_lp.hpp>
```

### Public Member Functions

- [BCP\\_lp\\_statistics](#) ()  
*The constructor just zeros out every timing data.*
- void [display](#) () const  
*Print out the statistics.*
- void [add](#) (const [BCP\\_lp\\_statistics](#) &stat)  
*Add the argument statistics to this one.*

### Packing and unpacking

- void **pack** ([BCP\\_buffer](#) &buf)
- void **unpack** ([BCP\\_buffer](#) &buf)

### 4.22.1 Detailed Description

NO OLD DOC.

Definition at line 56 of file BCP\_lp.hpp.

### 4.22.2 Member Function Documentation

#### 4.22.2.1 void BCP\_lp\_statistics::add ( const BCP\_lp\_statistics & stat )

Add the argument statistics to this one.

This method is used when multiple LP processes are running and their stats need to be combined.

The documentation for this class was generated from the following file:

- BCP\_lp.hpp

## 4.23 BCP\_lp\_user Class Reference

The [BCP\\_lp\\_user](#) class is the base class from which the user can derive a problem specific class to be used in the LP process.

```
#include <BCP_lp_user.hpp>
```

Inheritance diagram for BCP\_lp\_user:

Collaboration diagram for BCP\_lp\_user:

## Public Member Functions

- void [print](#) (const bool ifprint, const char \*format,...) const  
*A method to print a message with the process id.*
- int [process\\_id](#) () const  
*What is the process id of the current process.*
- int [parent](#) () const  
*the process id of the parent*
- void [send\\_message](#) (const int target, const [BCP\\_buffer](#) &buf, BCP\_message\_tag tag=BCP\_Msg\_User)  
*Send a message to a particular process.*
- void [receive\\_message](#) (const int sender, [BCP\\_buffer](#) &buf, BCP\_message\_tag tag=BCP\_Msg\_User)  
*Wait for a message and receive it.*
- void [broadcast\\_message](#) (const BCP\_process\_t proc\_type, const [BCP\\_buffer](#) &buf)  
*Broadcast the message to all processes of the given type.*
- virtual void [process\\_message](#) ([BCP\\_buffer](#) &buf)  
*Process a message that has been sent by another process' user part to this process' user part.*
- virtual [OsiSolverInterface](#) \* [initialize\\_solver\\_interface](#) ()  
*Create LP solver environment.*
- virtual void [initialize\\_int\\_and\\_sos\\_list](#) (std::vector< [OsiObject](#) \* > &intAndSosObjects)  
*Create the list of objects that can be used for branching (simple integer vars and SOS sets).*
- virtual void [initialize\\_new\\_search\\_tree\\_node](#) (const [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars, const [BCP\\_vec](#)< [BCP\\_cut](#) \* > &cuts, const [BCP\\_vec](#)< BCP\_obj\_status > &var\_status, const [BCP\\_vec](#)< BCP\_obj\_status > &cut\_status, [BCP\\_vec](#)< int > &var\_changed\_pos, [BCP\\_vec](#)< double > &var\_new\_bd, [BCP\\_vec](#)< int > &cut\_changed\_pos, [BCP\\_vec](#)< double > &cut\_new\_bd)  
*Initializing a new search tree node.*
- virtual void [load\\_problem](#) ([OsiSolverInterface](#) &osi, [BCP\\_problem\\_core](#) \*core, [BCP\\_var\\_set](#) &vars, [BCP\\_cut](#)←[\\_set](#) &cuts)  
*Load the problem specified by core, vars, and cuts into the solver interface.*
- virtual void [modify\\_lp\\_parameters](#) ([OsiSolverInterface](#) \*lp, const int changeType, bool in\_strong\_branching)  
*Modify parameters of the LP solver before optimization.*
- virtual void [process\\_lp\\_result](#) (const [BCP\\_lp\\_result](#) &lpres, const [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars, const [BCP\\_vec](#)←[\\_vec](#)< [BCP\\_cut](#) \* > &cuts, const double old\_lower\_bound, double &true\_lower\_bound, [BCP\\_solution](#) \*&sol, [BCP\\_vec](#)< [BCP\\_cut](#) \* > &new\_cuts, [BCP\\_vec](#)< [BCP\\_row](#) \* > &new\_rows, [BCP\\_vec](#)< [BCP\\_var](#) \* > &new←[\\_vars](#), [BCP\\_vec](#)< [BCP\\_col](#) \* > &new\_cols)  
*Process the result of an iteration.*
- virtual double [compute\\_lower\\_bound](#) (const double old\_lower\_bound, const [BCP\\_lp\\_result](#) &lpres, const [BCP\\_vec](#)←[\\_vec](#)< [BCP\\_var](#) \* > &vars, const [BCP\\_vec](#)< [BCP\\_cut](#) \* > &cuts)  
*Compute a true lower bound for the subproblem.*
- virtual [BCP\\_solution](#) \* [generate\\_heuristic\\_solution](#) (const [BCP\\_lp\\_result](#) &lpres, const [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars, const [BCP\\_vec](#)< [BCP\\_cut](#) \* > &cuts)  
*Try to generate a heuristic solution (or return one generated during cut/variable generation).*
- virtual void [restore\\_feasibility](#) (const [BCP\\_lp\\_result](#) &lpres, const std::vector< double \* > dual\_rays, const [BCP\\_vec](#)←[\\_vec](#)< [BCP\\_var](#) \* > &vars, const [BCP\\_vec](#)< [BCP\\_cut](#) \* > &cuts, [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars\_to\_add, [BCP\\_vec](#)< [BCP\\_col](#) \* > &cols\_to\_add)  
*Restoring feasibility.*
- void [reduced\\_cost\\_fixing](#) (const double \*dj, const double \*x, const double gap, [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars, int &newly\_changed)  
*Reduced cost fixing.*

- virtual BCP\_branching\_object\_relation [compare\\_branching\\_candidates](#) (BCP\_presolved\_lp\_brobj \*new\_solved, BCP\_presolved\_lp\_brobj \*old\_solved)  
*Decide which branching object is preferred for branching.*
- virtual void [set\\_actions\\_for\\_children](#) (BCP\_presolved\_lp\_brobj \*best)  
*Decide what to do with the children of the selected branching object.*
- virtual void [set\\_user\\_data\\_for\\_children](#) (BCP\_presolved\_lp\_brobj \*best, const int selected)  
*For each child create a user data object and put it into the appropriate entry in best->user\_data().*
- virtual void [set\\_user\\_data\\_for\\_children](#) (BCP\_presolved\_lp\_brobj \*best)  
*Deprecated version of the previous method (it does not pass the index of the selected branching candidate).*

### Methods to set and get the pointer to the BCP\_lp\_prob

object.

It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.

- void [setLpProblemPointer](#) (BCP\_lp\_prob \*ptr)  
*Set the pointer.*
- BCP\_lp\_prob \* [getLpProblemPointer](#) ()  
*Get the pointer.*

### Informational methods for the user.

- double [upper\\_bound](#) () const  
*Return what is the best known upper bound (might be BCP\_DBL\_MAX)*
- bool [over\\_ub](#) (double lb) const  
*Return true / false depending on whether the lb argument is over the current upper bound or not.*
- int [current\\_phase](#) () const  
*Return the phase the algorithm is in.*
- int [current\\_level](#) () const  
*Return the level of the search tree node being processed.*
- int [current\\_index](#) () const  
*Return the internal index of the search tree node being processed.*
- int [current\\_iteration](#) () const  
*Return the iteration count within the search tree node being processed.*
- double [start\\_time](#) () const  
*Return when the LP process started.*
- BCP\_user\_data \* [get\\_user\\_data](#) ()  
*Return a pointer to the BCP\_user\_data structure the user (may have) stored in this node.*

### Methods to get/set BCP parameters on the fly

- char [get\\_param](#) (const BCP\_lp\_par::chr\_params key) const
- int [get\\_param](#) (const BCP\_lp\_par::int\_params key) const
- double [get\\_param](#) (const BCP\_lp\_par::dbl\_params key) const
- const BCP\_string & [get\\_param](#) (const BCP\_lp\_par::str\_params key) const
- void [set\\_param](#) (const BCP\_lp\_par::chr\_params key, const bool val)
- void [set\\_param](#) (const BCP\_lp\_par::chr\_params key, const char val)
- void [set\\_param](#) (const BCP\_lp\_par::int\_params key, const int val)
- void [set\\_param](#) (const BCP\_lp\_par::dbl\_params key, const double val)
- void [set\\_param](#) (const BCP\_lp\_par::str\_params key, const char \*val)

### A methods to send a solution to the Tree Manager. The user can

invoke this method at any time to send off a solution.

- void **send\_feasible\_solution** (const [BCP\\_solution](#) \*sol)

### Constructor, Destructor

- **BCP\_lp\_user** ()
- virtual **~BCP\_lp\_user** ()

*Being virtual, the destructor invokes the destructor for the real type of the object being deleted.*

### Helper functions for selecting subset of entries from a double

*vector.*

*The indices (their position with respect to `first`) of the variables satisfying the criteria are returned in the last argument.*

- void **select\_nonzeros** (const double \*first, const double \*last, const double etol, [BCP\\_vec](#)< int > &nonzeros) const  
*Select all nonzero entries.*
- void **select\_zeros** (const double \*first, const double \*last, const double etol, [BCP\\_vec](#)< int > &zeros) const  
*Select all zero entries.*
- void **select\_positives** (const double \*first, const double \*last, const double etol, [BCP\\_vec](#)< int > &positives) const  
*Select all positive entries.*
- void **select\_fractions** (const double \*first, const double \*last, const double etol, [BCP\\_vec](#)< int > &fractions) const  
*Select all fractional entries.*

### Packing and unpacking methods

- virtual void **unpack\_module\_data** ([BCP\\_buffer](#) &buf)  
*Unpack the initial information sent to the LP process by the Tree Manager.*

### MIP feasibility testing of LP solutions and heuristics

- virtual [BCP\\_solution](#) \* **test\_feasibility** (const [BCP\\_lp\\_result](#) &lp\_result, const [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars, const [BCP\\_vec](#)< [BCP\\_cut](#) \* > &cuts)  
*Evaluate and return MIP feasibility of the current solution.*

*Helper functions for `test_feasibility`.*

*If the solution is feasible a pointer to a [BCP\\_solution\\_generic](#) object is returned. Note that the solutions generated by these helper functions **DO NOT OWN** the pointers in the `_vars` member of the solution. Also note that all of these functions assume that the specified integer tolerance is larger than the LP primal tolerance extracted from `lpres` and that the solution in `lpres` do not violate the bounds by more than the LP tolerance.*

- [BCP\\_solution\\_generic](#) \* **test\_binary** (const [BCP\\_lp\\_result](#) &lpres, const [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars, const double etol) const  
*Test whether all variables are 0/1.*
- [BCP\\_solution\\_generic](#) \* **test\_integral** (const [BCP\\_lp\\_result](#) &lpres, const [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars, const double etol) const  
*Test whether all variables are integer.*
- [BCP\\_solution\\_generic](#) \* **test\_full** (const [BCP\\_lp\\_result](#) &lpres, const [BCP\\_vec](#)< [BCP\\_var](#) \* > &vars, const double etol) const  
*Test whether the variables specified as integers are really integer.*



### Packing of solutions

- virtual void `pack_feasible_solution` (BCP\_buffer &buf, const BCP\_solution \*sol)  
*Pack a MIP feasible solution into a buffer.*
- virtual void `pack_primal_solution` (BCP\_buffer &buf, const BCP\_lp\_result &lp\_result, const BCP\_vec< BCP\_var \* > &vars, const BCP\_vec< BCP\_cut \* > &cuts)  
*Pack the information necessary for cut generation into the buffer.*
- virtual void `pack_dual_solution` (BCP\_buffer &buf, const BCP\_lp\_result &lp\_result, const BCP\_vec< BCP\_var \* > &vars, const BCP\_vec< BCP\_cut \* > &cuts)  
*Pack the information necessary for variable generation into the buffer.*

### Displaying of LP solutions

- virtual void `display_lp_solution` (const BCP\_lp\_result &lp\_result, const BCP\_vec< BCP\_var \* > &vars, const BCP\_vec< BCP\_cut \* > &cuts, const bool final\_lp\_solution)  
*Display the result of most recent LP optimization.*

### Converting cuts and variables into rows and columns

- virtual void `cuts_to_rows` (const BCP\_vec< BCP\_var \* > &vars, BCP\_vec< BCP\_cut \* > &cuts, BCP\_vec< BCP\_row \* > &rows, const BCP\_lp\_result &lpres, BCP\_object\_origin origin, bool allow\_multiple)  
*Convert (and possibly lift) a set of cuts into corresponding rows for the current LP relaxation.*
- virtual void `vars_to_cols` (const BCP\_vec< BCP\_cut \* > &cuts, BCP\_vec< BCP\_var \* > &vars, BCP\_vec< BCP\_col \* > &cols, const BCP\_lp\_result &lpres, BCP\_object\_origin origin, bool allow\_multiple)  
*Convert a set of variables into corresponding columns for the current LP relaxation.*

### Generating cuts and variables

- virtual void `generate_cuts_in_lp` (const BCP\_lp\_result &lpres, const BCP\_vec< BCP\_var \* > &vars, const BCP\_vec< BCP\_cut \* > &cuts, BCP\_vec< BCP\_cut \* > &new\_cuts, BCP\_vec< BCP\_row \* > &new\_rows)  
*Generate cuts within the LP process.*
- virtual void `generate_vars_in_lp` (const BCP\_lp\_result &lpres, const BCP\_vec< BCP\_var \* > &vars, const BCP\_vec< BCP\_cut \* > &cuts, const bool before\_fathom, BCP\_vec< BCP\_var \* > &new\_vars, BCP\_vec< BCP\_col \* > &new\_cols)  
*Generate variables within the LP process.*
- virtual BCP\_object\_compare\_result `compare_cuts` (const BCP\_cut \*c0, const BCP\_cut \*c1)  
*Compare two generated cuts.*
- virtual BCP\_object\_compare\_result `compare_vars` (const BCP\_var \*v0, const BCP\_var \*v1)  
*Compare two generated variables.*

### Logical fixing

- virtual void `logical_fixing` (const BCP\_lp\_result &lpres, const BCP\_vec< BCP\_var \* > &vars, const BCP\_vec< BCP\_cut \* > &cuts, const BCP\_vec< BCP\_obj\_status > &var\_status, const BCP\_vec< BCP\_obj\_status > &cut\_status, const int var\_bound\_changes\_since\_logical\_fixing, BCP\_vec< int > &changed\_pos, BCP\_vec< double > &new\_bd)  
*This method provides an opportunity for the user to tighten the bounds of variables.*

### Branching related methods

- virtual BCP\_branching\_decision `select_branching_candidates` (const BCP\_lp\_result &lpres, const BCP\_vec< BCP\_var \* > &vars, const BCP\_vec< BCP\_cut \* > &cuts, const BCP\_lp\_var\_pool &local\_var\_pool, const BCP\_lp\_cut\_pool &local\_cut\_pool, BCP\_vec< BCP\_lp\_branching\_object \* > &cands, bool force\_branch=false)  
*Decide whether to branch or not and select a set of branching candidates if branching is decided upon.*

### Helper functions for `select_branching_candidates()`

- virtual int `try_to_branch` (`OsiBranchingInformation` &branchInfo, `OsiSolverInterface` \*solver, `OsiChooseVariable` \*choose, `OsiBranchingObject` \*&branchObject, bool allowVarFix)
- void `branch_close_to_half` (const `BCP_lp_result` &lpres, const `BCP_vec`< `BCP_var` \* > &vars, const int to\_be\_selected, const double etol, `BCP_vec`< `BCP_lp_branching_object` \* > &candidates)  
*Select the "close-to-half" variables for strong branching.*
- void `branch_close_to_one` (const `BCP_lp_result` &lpres, const `BCP_vec`< `BCP_var` \* > &vars, const int to\_be\_selected, const double etol, `BCP_vec`< `BCP_lp_branching_object` \* > &candidates)  
*Select the "close-to-one" variables for strong branching.*
- void `append_branching_vars` (const double \*x, const `BCP_vec`< `BCP_var` \* > &vars, const `BCP_vec`< int > &select\_pos, `BCP_vec`< `BCP_lp_branching_object` \* > &candidates)  
*This helper method creates branching variable candidates and appends them to `cans`.*

### Purging the slack pool

- virtual void `purge_slack_pool` (const `BCP_vec`< `BCP_cut` \* > &slack\_pool, `BCP_vec`< int > &to\_be\_purged)  
*Selectively purge the list of slack cuts.*

## 4.23.1 Detailed Description

The `BCP_lp_user` class is the base class from which the user can derive a problem specific class to be used in the LP process.

In that derived class the user can store data to be used in the methods she overrides. Also that is the object the user must return in the `USER_initialize::lp_init()` method.

There are two kind of methods in the class. The non-virtual methods are helper functions for the built-in defaults, but the user can use them as well. The virtual methods execute steps in the BCP algorithm where the user might want to override the default behavior.

The default implementations fall into three major categories.

- Empty; doesn't do anything and immediately returns (e.g., `unpack_module_data()`).
- There is no reasonable default, so throw an exception. This happens if the parameter settings drive the flow of in a way that BCP can't perform the necessary function. This behavior is correct since such methods are invoked only if the parameter settings drive the flow of the algorithm that way, in which case the user better implement those methods
- A default is given. Frequently there are multiple defaults and parameters govern which one is selected (e.g., `test_feasibility()`).

Definition at line 75 of file `BCP_lp_user.hpp`.

## 4.23.2 Constructor & Destructor Documentation

### 4.23.2.1 virtual `BCP_lp_user::~BCP_lp_user` ( ) [inline], [virtual]

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 157 of file `BCP_lp_user.hpp`.

### 4.23.3 Member Function Documentation

#### 4.23.3.1 `bool BCP_lp_user::over_ub ( double lb ) const`

Return true / false depending on whether the *lb* argument is over the current upper bound or not.

#### 4.23.3.2 `void BCP_lp_user::select_nonzeros ( const double * first, const double * last, const double etol, BCP_vec< int > & nonzeros ) const`

Select all nonzero entries.

Those are considered nonzero that have absolute value greater than *etol*.

#### 4.23.3.3 `void BCP_lp_user::select_zeros ( const double * first, const double * last, const double etol, BCP_vec< int > & zeros ) const`

Select all zero entries.

Those are considered zero that have absolute value less than *etol*.

#### 4.23.3.4 `void BCP_lp_user::select_positives ( const double * first, const double * last, const double etol, BCP_vec< int > & positives ) const`

Select all positive entries.

Those are considered positive that have value greater than *etol*.

#### 4.23.3.5 `void BCP_lp_user::select_fractions ( const double * first, const double * last, const double etol, BCP_vec< int > & fractions ) const`

Select all fractional entries.

Those are considered fractional that are further than *etol* away from any integer value.

#### 4.23.3.6 `virtual void BCP_lp_user::unpack_module_data ( BCP_buffer & buf ) [virtual]`

Unpack the initial information sent to the LP process by the Tree Manager.

This information was packed by the method `BCP_tm_user::pack_module_data()` invoked with `BCP_ProcessType←_LP` as the third (target process type) argument.

Default: empty method.

#### 4.23.3.7 `virtual void BCP_lp_user::process_message ( BCP_buffer & buf ) [virtual]`

Process a message that has been sent by another process' user part to this process' user part.

#### 4.23.3.8 `virtual OsiSolverInterface* BCP_lp_user::initialize_solver_interface ( ) [virtual]`

Create LP solver environment.

Create the LP solver class that will be used for solving the LP relaxations. The default implementation picks up which `COIN_USE_XXX` is defined and initializes an lp solver of that type. This is probably OK for most users. The only reason to override this method is to be able to choose at runtime which lp solver to instantiate (maybe even different solvers on different processors). In this case she should probably also override the `pack_warmstart()` and `unpack_warmstart()` methods in this class and in the `BCP_tm_user` class.

**4.23.3.9** `virtual void BCP_lp_user::initialize_int_and_sos_list ( std::vector< OsiObject * > & intAndSosObjects )`  
[virtual]

Create the list of objects that can be used for branching (simple integer vars and SOS sets).

If nothing is done here then for each search tree node (just before starting to process the node) BCP will scan the variables and the matrix for candidates.

**4.23.3.10** `virtual void BCP_lp_user::initialize_new_search_tree_node ( const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts, const BCP_vec< BCP_obj_status > & var_status, const BCP_vec< BCP_obj_status > & cut_status, BCP_vec< int > & var_changed_pos, BCP_vec< double > & var_new_bd, BCP_vec< int > & cut_changed_pos, BCP_vec< double > & cut_new_bd )` [virtual]

Initializing a new search tree node.

This method serves as hook for the user to do some preprocessing on a search tree node before the node is processed. Also, logical fixing results can be returned in the last four parameters. This might be very useful if the branching implies significant tightening.

Default: empty method.

#### Parameters

<code>vars</code>	(IN) The variables in the current formulation
<code>cuts</code>	(IN) The cuts in the current formulation
<code>var_status</code>	(IN) The stati of the variables
<code>cut_status</code>	(IN) The stati of the cuts
<code>var_changed_pos</code>	(OUT) The positions of the variables whose bounds should be tightened
<code>var_new_bd</code>	(OUT) The new lb/ub of those variables
<code>cut_changed_pos</code>	(OUT) The positions of the cuts whose bounds should be tightened
<code>cut_new_bd</code>	(OUT) The new lb/ub of those cuts

**4.23.3.11** `virtual void BCP_lp_user::load_problem ( OsiSolverInterface & osi, BCP_problem_core * core, BCP_var_set & vars, BCP_cut_set & cuts )` [virtual]

Load the problem specified by core, vars, and cuts into the solver interface.

If the solver is an LP solver then the default is fine. If it's an NLP then the user has to do this herself.

**4.23.3.12** `virtual void BCP_lp_user::modify_lp_parameters ( OsiSolverInterface * lp, const int changeType, bool in_strong_branching )` [virtual]

Modify parameters of the LP solver before optimization.

This method provides an opportunity for the user to change parameters of the LP solver before optimization in the LP solver starts. The second argument indicates what has changed in the LP before this method is called. 0: no change; 1:

changes that affect primal feasibility (change in column/row bounds, added cuts); 2: changes that affect dual feasibility (added columns); 3: both. The last argument indicates whether the optimization is a "regular" optimization or it will take place in strong branching.

Default: If 1 or 2 then the appropriate simplex method will be hinted to the solver.

```
4.23.3.13 virtual void BCP_lp_user::process_lp_result ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars,
const BCP_vec< BCP_cut * > & cuts, const double old_lower_bound, double & true_lower_bound, BCP_solution
*& sol, BCP_vec< BCP_cut * > & new_cuts, BCP_vec< BCP_row * > & new_rows, BCP_vec< BCP_var *
> & new_vars, BCP_vec< BCP_col * > & new_cols ) [virtual]
```

Process the result of an iteration.

This includes:

- computing a true lower bound on the subproblem.  
In case column generation is done the lower bound for the subproblem might not be the same as the objective value of the current LP relaxation. Here the user has an option to return a true lower bound.
- test feasibility of the solution (or generate a heuristic solution)
- generating cuts and/or variables

The reason for the existence of this method is that (especially when column generation is done) these tasks are so intertwined that it is much easier to execute them in one method instead of in several separate methods.

The default behavior is to do nothing and invoke the individual methods one-by-one.

Parameters

<i>lp_result</i>	the result of the most recent LP optimization (IN)
<i>vars</i>	variables currently in the formulation (IN)
<i>cuts</i>	variables currently in the formulation (IN)
<i>old_lower_bound</i>	the previously known best lower bound (IN)
<i>new_cuts</i>	the vector of generated cuts (OUT)
<i>new_rows</i>	the correspondding rows(OUT)
<i>new_vars</i>	the vector of generated variables (OUT)
<i>new_cols</i>	the correspondding columns(OUT)

```
4.23.3.14 virtual double BCP_lp_user::compute_lower_bound ( const double old_lower_bound, const BCP_lp_result & lpres,
const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts ) [virtual]
```

Compute a true lower bound for the subproblem.

In case column generation is done the lower bound for the subproblem might not be the same as the objective value of the current LP relaxation. Here the user has an option to return a true lower bound.

The default implementation returns the objective value of the current LP relaxation if no column generation is done, otherwise returns the current (somehow previously computed) true lower bound.

```
4.23.3.15 virtual BCP_solution* BCP_lp_user::test_feasibility ( const BCP_lp_result & lp_result, const BCP_vec<
BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts ) [virtual]
```

Evaluate and return MIP feasibility of the current solution.

If the solution is MIP feasible, return a solution object otherwise return a NULL pointer. The user is also welcome to heuristically generate a solution and return a pointer to that solution (although the user will have another chance (after cuts and variables are generated) to return/create heuristically generated solutions. (After all, it's quite possible that solutions are generated during cut/variable generation.)

Default: test feasibility based on the `FeasibilityTest` parameter in `BCP_lp_par` which defaults to `BCP_FullTest_Feasible`.

#### Parameters

<i>lp_result</i>	the result of the most recent LP optimization
<i>vars</i>	variables currently in the formulation
<i>cuts</i>	variables currently in the formulation

**4.23.3.16** `BCP_solution_generic* BCP_lp_user::test_binary ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const double etol ) const`

Test whether all variables are 0/1.

Note that this method assumes that all variables are binary, i.e., their original lower/upper bounds are 0/1.

**4.23.3.17** `BCP_solution_generic* BCP_lp_user::test_integral ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const double etol ) const`

Test whether all variables are integer.

Note that this method assumes that all variables are integer.

**4.23.3.18** `BCP_solution_generic* BCP_lp_user::test_full ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const double etol ) const`

Test whether the variables specified as integers are really integer.

**4.23.3.19** `virtual BCP_solution* BCP_lp_user::generate_heuristic_solution ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts ) [virtual]`

Try to generate a heuristic solution (or return one generated during cut/variable generation).

Return a pointer to the generated solution or return a NULL pointer.

Default: Return a NULL pointer

**4.23.3.20** `virtual void BCP_lp_user::pack_feasible_solution ( BCP_buffer & buf, const BCP_solution * sol ) [virtual]`

Pack a MIP feasible solution into a buffer.

The solution will be unpacked in the Tree Manager by the `BCP_tm_user::unpack_feasible_solution()` method.

Default: The default implementation assumes that `sol` is a `BCP_solution_generic` object (containing variables at nonzero level) and packs it.

## Parameters

<i>buf</i>	(OUT) the buffer to pack into
<i>sol</i>	(IN) the solution to be packed

4.23.3.21 `virtual void BCP_lp_user::pack_primal_solution ( BCP_buffer & buf, const BCP_lp_result & lp_result, const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts ) [virtual]`

Pack the information necessary for cut generation into the buffer.

Note that the name of the method is `pack_primal_solution` because most likely that (or some part of that) will be needed for cut generation. However, if the user overrides the method she is free to pack anything (of course she'll have to unpack it in CG).

This information will be sent to the Cut Generator (and possibly to the Cut Pool) where the user has to unpack it. If the user uses the built-in method here, then the built-in method will be used in the Cut Generator as well.

Default: The content of the message depends on the value of the `PrimalSolForCG` parameter in [BCP\\_lp\\_par](#). By default the variables at nonzero level are packed.

## Parameters

<i>buf</i>	(OUT) the buffer to pack into
<i>lp_result</i>	(IN) the result of the most recent LP optimization
<i>vars</i>	(IN) variables currently in the formulation
<i>cuts</i>	(IN) cuts currently in the formulation

4.23.3.22 `virtual void BCP_lp_user::pack_dual_solution ( BCP_buffer & buf, const BCP_lp_result & lp_result, const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts ) [virtual]`

Pack the information necessary for variable generation into the buffer.

Note that the name of the method is `pack_dual_solution` because most likely that (or some part of that) will be needed for variable generation. However, if the user overrides the method she is free to pack anything (of course she'll have to unpack it in CG).

This information will be sent to the Variable Generator (and possibly to the Variable Pool) where the user has to unpack it. If the user uses the built-in method here, then the built-in method will be used in the Variable Generator as well.

Default: The content of the message depends on the value of the `DualSolForVG` parameter in [BCP\\_lp\\_par](#). By default the full dual solution is packed.

## Parameters

<i>buf</i>	(OUT) the buffer to pack into
<i>lp_result</i>	(IN) the result of the most recent LP optimization
<i>vars</i>	(IN) variables currently in the formulation
<i>cuts</i>	(IN) cuts currently in the formulation

4.23.3.23 `virtual void BCP_lp_user::display_lp_solution ( const BCP_lp_result & lp_result, const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts, const bool final_lp_solution ) [virtual]`

Display the result of most recent LP optimization.

This method is invoked every time an LP relaxation is optimized and the user can display (or not display) it.

Note that this method is invoked only if `final_lp_solution` is true (i.e., no cuts/variables were found) and the `LpVerb_FinalRelaxedSolution` parameter of `BCP_lp_par` is set to true (or alternatively, `final_lp_solution` is false and `LpVerb_RelaxedSolution` is true).

Default: display the solution if the appropriate verbosity code entry is set.

#### Parameters

<i>lp_result</i>	(IN) the result of the most recent LP optimization
<i>vars</i>	(IN) variables currently in the formulation
<i>final_lp_solution</i>	(IN) whether the lp solution is final or not.

```
4.23.3.24 virtual void BCP_lp_user::restore_feasibility ( const BCP_lp_result & lpres, const std::vector< double * > dual_rays,
const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts, BCP_vec< BCP_var * > &
vars_to_add, BCP_vec< BCP_col * > & cols_to_add ) [virtual]
```

Restoring feasibility.

This method is invoked before fathoming a search tree node that has been found infeasible *and* the variable pricing did not generate any new variables.

```
4.23.3.25 virtual void BCP_lp_user::cuts_to_rows ( const BCP_vec< BCP_var * > & vars, BCP_vec< BCP_cut * > & cuts,
BCP_vec< BCP_row * > & rows, const BCP_lp_result & lpres, BCP_object_origin origin, bool allow_multiple )
[virtual]
```

Convert (and possibly lift) a set of cuts into corresponding rows for the current LP relaxation.

Converting means computing for each cut the coefficients corresponding to each variable and creating `BCP_row` objects that can be added to the formulation.

This method has different purposes depending on the value of the last argument. If multiple expansion is not allowed then the user must generate a unique row for each cut. This unique row must always be the same for any given cut. This kind of operation is needed so that an LP relaxation can be exactly recreated.

On the other hand if multiple expansion is allowed then the user has (almost) free reign over what she returns. She can delete some of the `cuts` or append new ones (e.g., lifted ones) to the end. The result of the LP relaxation and the origin of the cuts are there to help her to make a decision about what to do. For example, she might want to lift cuts coming from the Cut Generator, but not those coming from the Cut Pool. The only requirement is that when this method returns the number of cuts and rows must be the same and the *i*-th row must be the unique row corresponding to the *i*-th cut.

#### Parameters

<i>vars</i>	the variables currently in the relaxation (IN)
<i>cuts</i>	the cuts to be converted (IN/OUT)
<i>rows</i>	the rows into which the cuts are converted (OUT)
<i>lpres</i>	solution to the current LP relaxation (IN)
<i>origin</i>	where the cuts come from (IN)
<i>allow_multiple</i>	whether multiple expansion, i.e., lifting, is allowed (IN)

Default: throw an exception (if this method is invoked then the user must have generated cuts and BCP has no way to know how to convert them).



4.23.3.26 `virtual void BCP_lp_user::vars_to_cols ( const BCP_vec< BCP_cut * > & cuts, BCP_vec< BCP_var * > & vars, BCP_vec< BCP_col * > & cols, const BCP_lp_result & lpres, BCP_object_origin origin, bool allow_multiple ) [virtual]`

Convert a set of variables into corresponding columns for the current LP relaxation.

Converting means to compute for each variable the coefficients corresponding to each cut and create `BCP_col` objects that can be added to the formulation.

See the documentation of `cuts_to_rows()` above for the use of this method (just reverse the role of cuts and variables.)

#### Parameters

<i>cuts</i>	the cuts currently in the relaxation (IN)
<i>vars</i>	the variables to be converted (IN/OUT)
<i>cols</i>	the columns the variables convert into (OUT)
<i>lpres</i>	solution to the current LP relaxation (IN)
<i>origin</i>	where the do the cuts come from (IN)
<i>allow_multiple</i>	whether multiple expansion, i.e., lifting, is allowed (IN)

Default: throw an exception (if this method is invoked then the user must have generated variables and BCP has no way to know how to convert them).

4.23.3.27 `virtual void BCP_lp_user::generate_cuts_in_lp ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts, BCP_vec< BCP_cut * > & new_cuts, BCP_vec< BCP_row * > & new_rows ) [virtual]`

Generate cuts within the LP process.

Sometimes too much information would need to be transmitted for cut generation (e.g., the full tableau for Gomory cuts) or the cut generation is so fast that transmitting the info would take longer than generating the cuts. In such cases it might better to generate the cuts locally. This routine provides the opportunity.

Default: empty for now. To be interfaced to Cgl.

#### Parameters

<i>lpres</i>	solution to the current LP relaxation (IN)
<i>vars</i>	the variables currently in the relaxation (IN)
<i>cuts</i>	the cuts currently in the relaxation (IN)
<i>new_cuts</i>	the vector of generated cuts (OUT)
<i>new_rows</i>	the corresponding rows (OUT)

4.23.3.28 `virtual void BCP_lp_user::generate_vars_in_lp ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts, const bool before_fathom, BCP_vec< BCP_var * > & new_vars, BCP_vec< BCP_col * > & new_cols ) [virtual]`

Generate variables within the LP process.

Sometimes too much information would need to be transmitted for variable generation or the variable generation is so fast that transmitting the info would take longer than generating the variables. In such cases it might be better to generate the variables locally. This routine provides the opportunity.

Default: empty method.

## Parameters

<i>lpres</i>	solution to the current LP relaxation (IN)
<i>vars</i>	the variables currently in the relaxation (IN)
<i>cuts</i>	the cuts currently in the relaxation (IN)
<i>before_fathom</i>	if true then BCP is about to fathom the node, so spend some extra effort generating variables if you want to avoid that...
<i>new_vars</i>	the vector of generated variables (OUT)
<i>new_cols</i>	the corresponding columns (OUT)

4.23.3.29 `virtual BCP_object_compare_result BCP_lp_user::compare_cuts ( const BCP_cut * c0, const BCP_cut * c1 )`  
`[virtual]`

Compare two generated cuts.

Cuts are generated in different iterations, they come from the Cut Pool, etc. There is a very real possibility that the LP process receives several cuts that are either identical or one of them is better than another (cuts off everything the other cuts off). This routine is used to decide which one to keep if not both.

Default: Return `BCP_DifferentObjs`.

4.23.3.30 `virtual BCP_object_compare_result BCP_lp_user::compare_vars ( const BCP_var * v0, const BCP_var * v1 )`  
`[virtual]`

Compare two generated variables.

Variables are generated in different iterations, they come from the Variable Pool, etc. There is a very real possibility that the LP process receives several variables that are either identical or one of them is better than another (e.g., almost identical but has much lower reduced cost). This routine is used to decide which one to keep if not both.

Default: Return `BCP_DifferentObjs`.

4.23.3.31 `virtual void BCP_lp_user::logical_fixing ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts, const BCP_vec< BCP_obj_status > & var_status, const BCP_vec< BCP_obj_status > & cut_status, const int var_bound_changes_since_logical_fixing, BCP_vec< int > & changed_pos, BCP_vec< double > & new_bd )` `[virtual]`

This method provides an opportunity for the user to tighten the bounds of variables.

The method is invoked after reduced cost fixing. The results are returned in the last two parameters.

Default: empty method.

## Parameters

<i>lpres</i>	the result of the most recent LP optimization,
<i>vars</i>	the variables in the current formulation,
<i>status</i>	the status of the variables as known to the system,
<i>var_bound_↔ changes_since↔ _logical_fixing</i>	the number of variables whose bounds have changed (by reduced cost fixing) since the most recent invocation of this method that has actually forced changes returned something in the last two arguments,

<i>changed_pos</i>	the positions of the variables whose bounds should be changed
<i>new_bd</i>	the new bounds (lb/ub pairs) of these variables.

4.23.3.32 `void BCP_lp_user::reduced_cost_fixing ( const double * dj, const double * x, const double gap, BCP_vec< BCP_var * > & vars, int & newly_changed )`

Reduced cost fixing.

This is not exactly a helper function, but the user might want to invoke it...

4.23.3.33 `virtual BCP_branching_decision BCP_lp_user::select_branching_candidates ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const BCP_vec< BCP_cut * > & cuts, const BCP_lp_var_pool & local_var_pool, const BCP_lp_cut_pool & local_cut_pool, BCP_vec< BCP_lp_branching_object * > & cands, bool force_branch = false ) [virtual]`

Decide whether to branch or not and select a set of branching candidates if branching is decided upon.

The return value indicates what should be done: branching, continuing with the same node or abandoning the node completely.

Default: Branch if both local pools are empty. If branching is done then several (based on the `StrongBranch_CloseToHalfNum` and `StrongBranch_CloseToOneNum` parameters in `BCP_lp_par`) variables are selected for strong branching.

"Close-to-half" variables are those that should be integer and are at a fractional level. The measure of their fractionality is their distance from the closest integer. The most fractional variables will be selected, i.e., those that are close to half. If there are too many such variables then those with higher objective value have priority.

"Close-to-on" is interpreted in a more literal sense. It should be used only if the integer variables are binary as it select those fractional variables which are away from 1 but are still close. If there are too many such variables then those with lower objective value have priority.

#### Parameters

<i>lpres</i>	the result of the most recent LP optimization.
<i>vars</i>	the variables in the current formulation.
<i>cuts</i>	the cuts in the current formulation.
<i>local_var_pool</i>	the local pool that holds variables with negative reduced cost. In case of continuing with the node the best so many variables will be added to the formulation (those with the most negative reduced cost).
<i>local_cut_pool</i>	the local pool that holds violated cuts. In case of continuing with the node the best so many cuts will be added to the formulation (the most violated ones).
<i>cands</i>	the generated branching candidates.
<i>force_branch</i>	indicate whether to force branching regardless of the size of the local cut/var pools

4.23.3.34 `void BCP_lp_user::branch_close_to_half ( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars, const int to_be_selected, const double etol, BCP_vec< BCP_lp_branching_object * > & candidates )`

Select the "close-to-half" variables for strong branching.

Variables that are at least `etol` away from integrality are considered and `to_be_selected` of them will be picked up.

```
4.23.3.35 void BCP_lp_user::branch_close_to_one( const BCP_lp_result & lpres, const BCP_vec< BCP_var * > & vars,
const int to_be_selected, const double etol, BCP_vec< BCP_lp_branching_object * > & candidates )
```

Select the "close-to-one" variables for strong branching.

Variables that are at least `etol` away from integrality are considered and `to_be_selected` of them will be picked up.

```
4.23.3.36 void BCP_lp_user::append_branching_vars( const double * x, const BCP_vec< BCP_var * > & vars, const
BCP_vec< int > & select_pos, BCP_vec< BCP_lp_branching_object * > & candidates )
```

This helper method creates branching variable candidates and appends them to `cans`.

The indices (in the current formulation) of the variables from which candidates should be created are listed in `select_pos`.

```
4.23.3.37 virtual BCP_branching_object_relation BCP_lp_user::compare_branching_candidates( BCP_presolved_lp_brobj *
new_solved, BCP_presolved_lp_brobj * old_solved ) [virtual]
```

Decide which branching object is preferred for branching.

Based on the member fields of the two presolved candidate branching objects decide which one should be preferred for really branching on it. Possible return values are: `BCP_OldPresolvedIsBetter`, `BCP_NewPresolvedIsBetter` and `BCP_NewPresolvedIsBetter_BranchOnIt`. This last value (besides specifying which candidate is preferred) also indicates that no further candidates should be examined, branching should be done on this candidate.

Default: The behavior of this method is governed by the `BranchingObjectComparison` parameter in [BCP\\_lp\\_par](#).

```
4.23.3.38 virtual void BCP_lp_user::set_actions_for_children( BCP_presolved_lp_brobj * best ) [virtual]
```

Decide what to do with the children of the selected branching object.

Fill out the `_child_action` field in `best`. This will specify for every child what to do with it. Possible values for each individual child are `BCP_FathomChild`, `BCP_ReturnChild` and `BCP_KeepChild`. There can be at most child with this last action specified. It means that in case of diving this child will be processed by this LP process as the next search tree node.

Default: Every action is `BCP_ReturnChild`. However, if BCP dives then one child will be mark with `BCP_KeepChild`. The decision which child to keep is based on the `ChildPreference` parameter in [BCP\\_lp\\_par](#). Also, if a child has a presolved lower bound that is higher than the current upper bound then that child is mark as `BCP_FathomChild`.

THINK\*: Should those children be sent back for processing in the next phase?

```
4.23.3.39 virtual void BCP_lp_user::set_user_data_for_children( BCP_presolved_lp_brobj * best, const int selected )
[virtual]
```

For each child create a user data object and put it into the appropriate entry in `best->user_data()`.

When this function is called the `best->user_data()` vector is already the right size and is filled with 0 pointers. The second argument is usefull if strong branching was done. It is the index of the branching candidate that was selected for branching (the one that's the source of `best`).

4.23.3.40 `virtual void BCP_lp_user::purge_slack_pool ( const BCP_vec< BCP_cut * > & slack_pool, BCP_vec< int > & to_be_purged ) [virtual]`

Selectively purge the list of slack cuts.

When a cut becomes ineffective and is eventually purged from the LP formulation it is moved into `slack_pool`. The user might consider cuts might later for branching. This function enables the user to purge any cut from the slack pool (those she wouldn't consider anyway). Of course, the user is not restricted to these cuts when branching, this is only there to help to collect slack cuts. The user should put the indices of the cuts to be purged into the provided vector.

Default: Purges the slack cut pool according to the `SlackCutDiscardingStrategy` rule in [BCP\\_lp\\_par](#) (purge everything before every iteration or before a new search tree node).

#### Parameters

<code>slack_pool</code>	the pool of slacks. (IN)
<code>to_be_purged</code>	the indices of the cuts to be purged. (OUT)

The documentation for this class was generated from the following file:

- `BCP_lp_user.hpp`

## 4.24 BCP\_lp\_var\_pool Class Reference

Inheritance diagram for `BCP_lp_var_pool`:

Collaboration diagram for `BCP_lp_var_pool`:

### Additional Inherited Members

#### 4.24.1 Detailed Description

Definition at line 109 of file `BCP_lp_pool.hpp`.

The documentation for this class was generated from the following file:

- `BCP_lp_pool.hpp`

## 4.25 BCP\_lp\_waiting\_col Class Reference

#### 4.25.1 Detailed Description

Definition at line 78 of file `BCP_lp_pool.hpp`.

The documentation for this class was generated from the following file:

- `BCP_lp_pool.hpp`

## 4.26 BCP\_lp\_waiting\_row Class Reference

### 4.26.1 Detailed Description

Definition at line 15 of file BCP\_lp\_pool.hpp.

The documentation for this class was generated from the following file:

- BCP\_lp\_pool.hpp

## 4.27 BCP\_MemPool Class Reference

### 4.27.1 Detailed Description

Definition at line 8 of file BCP\_mempool.hpp.

The documentation for this class was generated from the following file:

- BCP\_mempool.hpp

## 4.28 BCP\_message\_environment Class Reference

This is an abstract base class that describes the message passing environment.

```
#include <BCP_message.hpp>
```

Inheritance diagram for BCP\_message\_environment:

### Public Member Functions

#### Destructor

- virtual [~BCP\\_message\\_environment](#) ()  
*Being virtual, the destructor invokes the destructor for the real type of the object being deleted.*

#### Registering a process

- virtual int [register\\_process](#) (USER\_initialize \*user\_init)=0  
*A process can register (receive its process id) with the message passing environment.*

#### Identify parent process

- virtual int [parent\\_process](#) ()=0  
*Return the process id of the parent process (the process that spawned the current process).*

#### Testing processes

- virtual bool [alive](#) (const int pid)=0  
*Test if the process given by the argument is alive or not.*
- virtual const int \* [alive](#) (int num, const int \*pids)=0  
*Test if the processes given by the process array in the argument are alive or not.*

#### Send to one process

- virtual void [send](#) (const int target, const BCP\_message\_tag tag)=0  
*Send an empty message (message tag only) to the process given by the first argument.*
- virtual void [send](#) (const int target, const BCP\_message\_tag tag, const [BCP\\_buffer](#) &buf)=0  
*Send the message in the buffer with the given message tag to the process given by the first argument.*

### Broadcasting

- virtual void [multicast](#) (int num, const int \*targets, const BCP\_message\_tag tag)=0  
*Send an empty message (message tag only) to all the processes in the process array.*
- virtual void [multicast](#) (int num, const int \*targets, const BCP\_message\_tag tag, const [BCP\\_buffer](#) &buf)=0  
*Send the message in the buffer with the given message tag to all processes in the process array.*

### Receiving

- virtual void [receive](#) (const int source, const BCP\_message\_tag tag, [BCP\\_buffer](#) &buf, const double timeout)=0  
*Blocking receive with timeout.*
- virtual bool [probe](#) (const int source, const BCP\_message\_tag tag)=0  
*Probe if there are any messages from the given process with the given message tag.*

### Starting a process

- virtual int [start\\_process](#) (const [BCP\\_string](#) &exe, const bool debug)=0  
*Spawn a new process.*
- virtual int [start\\_process](#) (const [BCP\\_string](#) &exe, const [BCP\\_string](#) &machine, const bool debug)=0  
*Spawn a new process on the machine specified by the second argument.*
- virtual bool [start\\_processes](#) (const [BCP\\_string](#) &exe, const int proc\_num, const bool debug, int \*ids)=0  
*Spawn *proc\_num* processes, all with the same executable.*
- virtual bool [start\\_processes](#) (const [BCP\\_string](#) &exe, const int proc\_num, const [BCP\\_vec](#)< [BCP\\_string](#) > &machines, const bool debug, int \*ids)=0  
*Spawn *proc\_num* processes on the machines given by the third argument, all with the same executable.*

### Additional function for MPI interface

- virtual int [num\\_procs](#) ()  
*Return the number of processes.*

## 4.28.1 Detailed Description

This is an abstract base class that describes the message passing environment.

The implementation of the message passing protocol must implement this class.

All methods are pure virtual, enforcing the correct overriding of the methods.

Definition at line 30 of file BCP\_message.hpp.

## 4.28.2 Constructor & Destructor Documentation

### 4.28.2.1 virtual BCP\_message\_environment::~~BCP\_message\_environment ( ) [inline],[virtual]

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 36 of file BCP\_message.hpp.

### 4.28.3 Member Function Documentation

4.28.3.1 `virtual int BCP_message_environment::register_process ( USER_initialize * user_init )` [pure virtual]

A process can register (receive its process id) with the message passing environment.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.2 `virtual int BCP_message_environment::parent_process ( )` [pure virtual]

Return the process id of the parent process (the process that spawned the current process).

Returns null if the process does not have a parent.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.3 `virtual bool BCP_message_environment::alive ( const int pid )` [pure virtual]

Test if the process given by the argument is alive or not.

Return true if alive, false otherwise.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.4 `virtual const int* BCP_message_environment::alive ( int num, const int * pids )` [pure virtual]

Test if the processes given by the process array in the argument are alive or not.

Return a pointer to the first dead process, or to the end of the array if there are no dead processes.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.5 `virtual void BCP_message_environment::send ( const int target, const BCP_message_tag tag )` [pure virtual]

Send an empty message (message tag only) to the process given by the first argument.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.6 `virtual void BCP_message_environment::send ( const int target, const BCP_message_tag tag, const BCP_buffer & buf )`  
[pure virtual]

Send the message in the buffer with the given message tag to the process given by the first argument.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.7 `virtual void BCP_message_environment::multicast ( int num, const int * targets, const BCP_message_tag tag )` [pure virtual]

Send an empty message (message tag only) to all the processes in the process array.

Implemented in [BCP\\_single\\_environment](#).



4.28.3.8 `virtual void BCP_message_environment::multicast ( int num, const int * targets, const BCP_message_tag tag, const BCP_buffer & buf ) [pure virtual]`

Send the message in the buffer with the given message tag to all processes in the process array.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.9 `virtual void BCP_message_environment::receive ( const int source, const BCP_message_tag tag, BCP_buffer & buf, const double timeout ) [pure virtual]`

Blocking receive with timeout.

Wait until a message is received from the given process with the given message tag within the given timelimit. If `timeout` is positive then the receive routine times out after this many seconds. If it is negative then the method blocks until a message is received. The received message is saved into the buffer. With a 0 pointer as the first argument (or the predefined `BCP_AnyProcess` process id can be used) a message from any process will be accepted. Messages with any message tag are accepted if the second argument is `BCP_Msg_AnyMessage`.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.10 `virtual bool BCP_message_environment::probe ( const int source, const BCP_message_tag tag ) [pure virtual]`

Probe if there are any messages from the given process with the given message tag.

Return true if such a message is found, false otherwise. Note that the message is not "read", only its existence is checked. Similarly as above, the wild cards `BCP_AnyProcess` and `BCP_Msg_AnyMessage` can be used.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.11 `virtual int BCP_message_environment::start_process ( const BCP_string & exe, const bool debug ) [pure virtual]`

Spawn a new process.

The first argument contains the path to the executable to be spawned. If the second argument is set to true, then the executable is spawned under a debugger.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.12 `virtual int BCP_message_environment::start_process ( const BCP_string & exe, const BCP_string & machine, const bool debug ) [pure virtual]`

Spawn a new process on the machine specified by the second argument.

Implemented in [BCP\\_single\\_environment](#).

4.28.3.13 `virtual bool BCP_message_environment::start_processes ( const BCP_string & exe, const int proc_num, const bool debug, int * ids ) [pure virtual]`

Spawn `proc_num` processes, all with the same executable.

NOTE: `ids` must be allocated already and have enough space for `proc_num` entries.

**Returns**

true/false depending on success.

Implemented in [BCP\\_single\\_environment](#).

```
4.28.3.14 virtual bool BCP_message_environment::start_processes ( const BCP_string & exe, const int proc_num, const
BCP_vec< BCP_string > & machines, const bool debug, int* ids ) [pure virtual]
```

Spawn `proc_num` processes on the machines given by the third argument, all with the same executable.

NOTE: `ids` must be allocated already and have enough space for `proc_num` entries.

**Returns**

true/false depending on success.

Implemented in [BCP\\_single\\_environment](#).

```
4.28.3.15 virtual int BCP_message_environment::num_procs ( ) [inline],[virtual]
```

Return the number of processes.

For non-MPI this just returns 0.

Definition at line 151 of file `BCP_message.hpp`.

The documentation for this class was generated from the following file:

- `BCP_message.hpp`

## 4.29 BCP\_node\_change Class Reference

Inheritance diagram for `BCP_node_change`:

Collaboration diagram for `BCP_node_change`:

### 4.29.1 Detailed Description

Definition at line 19 of file `BCP_node_change.hpp`.

The documentation for this class was generated from the following file:

- `BCP_node_change.hpp`

## 4.30 BCP\_node\_storage\_in\_tm Struct Reference

NO OLD DOC.

```
#include <BCP_lp_node.hpp>
```

### 4.30.1 Detailed Description

NO OLD DOC.

This structure describes the ways various pieces of the current node's description are stored in the Tree Manager.

Definition at line 25 of file BCP\_lp\_node.hpp.

The documentation for this struct was generated from the following file:

- BCP\_lp\_node.hpp

## 4.31 BCP\_obj\_change Class Reference

### 4.31.1 Detailed Description

Definition at line 23 of file BCP\_obj\_change.hpp.

The documentation for this class was generated from the following file:

- BCP\_obj\_change.hpp

## 4.32 BCP\_obj\_set\_change Class Reference

This class stores data about how an object set (set of vars or set of cuts) changes.

```
#include <BCP_obj_change.hpp>
```

Collaboration diagram for BCP\_obj\_set\_change:

### 4.32.1 Detailed Description

This class stores data about how an object set (set of vars or set of cuts) changes.

Definition at line 57 of file BCP\_obj\_change.hpp.

The documentation for this class was generated from the following file:

- BCP\_obj\_change.hpp

## 4.33 BCP\_parameter Class Reference

This parameter indeintifies a single parameter entry.

```
#include <BCP_parameters.hpp>
```

### Public Member Functions

#### Constructors / Destructor

- [BCP\\_parameter](#) ()

*The default constructor creates a phony parameter.*

- [BCP\\_parameter](#) (const BCP\_parameter\_t t, const int i)  
*Constructor where members are specified.*
- [~BCP\\_parameter](#) ()  
*The destructor.*

#### Query methods

- BCP\_parameter\_t [type](#) () const  
*Return the type of the parameter.*
- int [index](#) () const  
*Return the index of the parameter within all parameters of the same type.*

### 4.33.1 Detailed Description

This parameter indeintifies a single parameter entry.

Definition at line 52 of file BCP\_parameters.hpp.

### 4.33.2 Constructor & Destructor Documentation

#### 4.33.2.1 BCP\_parameter::BCP\_parameter ( ) [inline]

The default constructor creates a phony parameter.

Definition at line 68 of file BCP\_parameters.hpp.

#### 4.33.2.2 BCP\_parameter::BCP\_parameter ( const BCP\_parameter\_t t, const int i ) [inline]

Constructor where members are specified.

Definition at line 70 of file BCP\_parameters.hpp.

#### 4.33.2.3 BCP\_parameter::~~BCP\_parameter ( ) [inline]

The destructor.

Definition at line 73 of file BCP\_parameters.hpp.

### 4.33.3 Member Function Documentation

#### 4.33.3.1 BCP\_parameter\_t BCP\_parameter::type ( ) const [inline]

Return the type of the parameter.

Definition at line 79 of file BCP\_parameters.hpp.

#### 4.33.3.2 int BCP\_parameter::index ( ) const [inline]

Return the index of the parameter within all parameters of the same type.

Definition at line 82 of file BCP\_parameters.hpp.

The documentation for this class was generated from the following file:

- BCP\_parameters.hpp

## 4.34 BCP\_parameter\_set< Par > Class Template Reference

This is the class serves as a holder for a set of parameters.

```
#include <BCP_parameters.hpp>
```

Inheritance diagram for BCP\_parameter\_set< Par >:

Collaboration diagram for BCP\_parameter\_set< Par >:

### Public Member Functions

#### Query methods

*The members of the parameter set can be queried for using the overloaded entry() method.*

*Using the example in the class documentation the user can get a parameter with the "`<code>param.entry(USER_par::parameter_name)</code>`" expression.*

- char **entry** (const chr\_params key) const
- int **entry** (const int\_params key) const
- double **entry** (const dbl\_params key) const
- const BCP\_string & **entry** (const str\_params key) const
- const BCP\_vec< BCP\_string > & **entry** (const str\_array\_params key) const

#### Set methods

*First, there is the assignment operator that sets the whole parameter set at once.*

*Individual members of the parameter set can be set for using the overloaded set\_entry() method. Using the example in the class documentation the user can set a parameter with the "`<code>param.set_entry(USER_par::parameter_name, param_value)</code>`" expression.*

- BCP\_parameter\_set< Par > & **operator=** (const BCP\_parameter\_set< Par > &x)
- void **set\_entry** (const chr\_params key, const char val)
- void **set\_entry** (const chr\_params key, const bool val)
- void **set\_entry** (const int\_params key, const int val)
- void **set\_entry** (const dbl\_params key, const double val)
- void **set\_entry** (const str\_params key, const char \*val)
- void **set\_entry** (const str\_array\_params key, const char \*val)
- void **set\_entry** (const BCP\_parameter key, const char \*val)

#### Read parameters from a stream.

- void **read\_from\_stream** (std::istream &parstream)  
*Read the parameters from the stream specified in the argument.*

#### Expand parameter value (look for environment vars).

- std::string **expand** (const char \*value)

#### Read parameters from a file.

- void **read\_from\_file** (const char \*paramfile)  
*Simply invoke reading from a stream.*

### Read parameters from the command line

- void `read_from_arglist` (const int argnum, const char \*const \*arglist)  
*Simply invoke reading from a stream.*

### Write parameters to a stream.

- void `write_to_stream` (std::ostream &ostream) const  
*Write keyword-value pairs to the stream specified in the argument.*

### Packing/unpacking methods

- void `pack` (BCP\_buffer &buf)  
*Pack the parameter set into the buffer.*
- void `unpack` (BCP\_buffer &buf)  
*Unpack the parameter set from the buffer.*

### Constructors and destructor.

- `BCP_parameter_set` ()  
*The default constructor creates a parameter set with from the template argument structure.*
- `~BCP_parameter_set` ()  
*The destructor deletes all data members.*

## 4.34.1 Detailed Description

```
template<class Par>class BCP_parameter_set< Par >
```

This is the class serves as a holder for a set of parameters.

For example, BCP stores has a parameter set for each process. Of course, the user can use this class for her own parameters. To use this class the user must

1. first create a `struct` with the names of the parameters (see, e.g., `BCP_tm_par`. Assume that the structure is named `USER_par`.
2. define the member functions `BCP_parameter_set<USER_par>::create_keyword_list()` and `BCP_parameter_set<USER_par>::set_default_entries()`. For an example look at the file `B←CP-common/TM/BCP_tm_param.cpp`. Essentially, the first method defines what keywords should be looked for in the parameter file, and if one is found which parameter should take the corresponding value; the other method specifies the default values for each parameter.
3. in her source code declare a variable "`<code>BCP_parameter_set<USER_par> param;</code>`"

After this the user can read in the parameters from a file, she can set/access the parameters in the parameter, etc.

Definition at line 113 of file `BCP_parameters.hpp`.

## 4.34.2 Constructor & Destructor Documentation

### 4.34.2.1 `template<class Par> BCP_parameter_set< Par >::BCP_parameter_set( ) [inline]`

The default constructor creates a parameter set with from the template argument structure.

The keyword list is created and the defaults are set.

Definition at line 564 of file `BCP_parameters.hpp`.

4.34.2.2 `template<class Par> BCP_parameter_set< Par >::~~BCP_parameter_set( )` `[inline]`

The destructor deletes all data members.

Definition at line 576 of file BCP\_parameters.hpp.

### 4.34.3 Member Function Documentation

4.34.3.1 `template<class Par> void BCP_parameter_set< Par >::read_from_stream( std::istream & parstream )`  
`[inline]`

Read the parameters from the stream specified in the argument.

The stream is interpreted as a lines separated by newline characters. The first word on each line is tested for match with the keywords specified in the `create_keyword_list()` method. If there is a match then the second word will be interpreted as the value for the corresponding parameter. Any further words on the line are discarded. Every non-matching line is discarded.

If the keyword corresponds to a non-array parameter then the new value simply overwrites the old one. Otherwise, i.e., if it is a `StringArrayPar`, the value is appended to the list of strings in that array.

Definition at line 262 of file BCP\_parameters.hpp.

4.34.3.2 `template<class Par> void BCP_parameter_set< Par >::read_from_file( const char * paramfile )` `[inline]`

Simply invoke reading from a stream.

Definition at line 441 of file BCP\_parameters.hpp.

4.34.3.3 `template<class Par> void BCP_parameter_set< Par >::read_from_arglist( const int argnum, const char *const * arglist )` `[inline]`

Simply invoke reading from a stream.

Definition at line 454 of file BCP\_parameters.hpp.

4.34.3.4 `template<class Par> void BCP_parameter_set< Par >::write_to_stream( std::ostream & outstream ) const`  
`[inline]`

Write keyword-value pairs to the stream specified in the argument.

Each keyword-value pair is separated by a newline character.

Definition at line 476 of file BCP\_parameters.hpp.

4.34.3.5 `template<class Par> void BCP_parameter_set< Par >::pack( BCP_buffer & buf )` `[inline]`

Pack the parameter set into the buffer.

Definition at line 522 of file BCP\_parameters.hpp.

4.34.3.6 `template<class Par> void BCP_parameter_set< Par >::unpack( BCP_buffer & buf )` `[inline]`

Unpack the parameter set from the buffer.

Definition at line 535 of file BCP\_parameters.hpp.

The documentation for this class was generated from the following file:

- BCP\_parameters.hpp

## 4.35 BCP\_presolved\_lp\_brobj Class Reference

A presolved branching object candidate.

```
#include <BCP_lp_branch.hpp>
```

### Public Member Functions

#### Constructor and destructor

- [BCP\\_presolved\\_lp\\_brobj](#) ([BCP\\_lp\\_branching\\_object](#) \**candidate*)  
*The only one way to construct a presolved branching object is to create it from a regular branching object.*
- [~BCP\\_presolved\\_lp\\_brobj](#) ()  
*The destructor simply deletes every member (deletes every lpres in the vector).*

#### Query methods

- [BCP\\_lp\\_branching\\_object](#) \* *candidate* ()  
*Return a pointer to the candidate.*
- const [BCP\\_lp\\_branching\\_object](#) \* *candidate* () const  
*Return a const pointer to the candidate.*
- const [BCP\\_lp\\_result](#) & *lpres* (const int *child\_ind*) const  
*Return a const reference to the presolved results of the *child\_ind*-th child.*
- [BCP\\_vec](#)< [BCP\\_child\\_action](#) > & *action* ()  
*Return a reference to the actions to be taken.*
- const [BCP\\_vec](#)< [BCP\\_child\\_action](#) > & *action* () const  
*Return a const reference to the actions to be taken.*
- [BCP\\_vec](#)< [BCP\\_user\\_data](#) \* > & *user\_data* ()  
*Return a reference to the user data vector.*
- const [BCP\\_vec](#)< [BCP\\_user\\_data](#) \* > & *user\_data* () const  
*Return a const reference to the user data vector.*
- bool *fathomable* (const double *objval\_limit*) const  
*Return true if every children can be fathomed.*
- bool *had\_numerical\_problems* () const  
*Return true if at least one child had numerical difficulties while presolving.*

#### Modifying methods

- void *initialize\_lower\_bound* (const double *val*)
- void *keep\_no\_child* ()
- bool *is\_pruned* () const
- void *get\_lower\_bounds* ([BCP\\_vec](#)< double > &*obj*)  
*Fill up *obj* with the lower bound on each child.*
- void *set\_lower\_bounds* (const [BCP\\_vec](#)< double > &*obj*)  
*Fill up the lower bounds on the children with the content of *obj*.*
- void *get\_results* ([OsiSolverInterface](#) &*lp*, const int *child\_ind*)  
*Extract the lp results from the LP solver for the *child\_ind*-th child.*



- void [fake\\_objective\\_values](#) (const double itlim\_objval)  
*Examine the termination codes for the children and for those that do not have a valid lower bound fake the objective value depending on the termination code:*
- void [set\\_objective\\_values](#) (const [BCP\\_vec](#)< double > &obj, const [BCP\\_vec](#)< int > &termcode, const double itlim\_objval)  
*Set the appropriate fields of all \_lpres to the given termcode and objval if the termcode is "normal".*
- void [swap](#) ([BCP\\_presolved\\_lp\\_brobj](#) &rhs)  
*swap the two presolved branching object*
- [BCP\\_vec](#)< [BCP\\_vec](#)< [BCP\\_cut](#) \* > > & [get\\_new\\_cuts](#) ()
- [BCP\\_vec](#)< [BCP\\_vec](#)< [BCP\\_row](#) \* > > & [get\\_new\\_rows](#) ()

### 4.35.1 Detailed Description

A presolved branching object candidate.

During strong branching all children of a branching object (of class [[BCP\\_lp\\_branching\\_object](#)]{[BCP\\_lp\\_branching\\_object.html](#)}) are *presolved* to gain some insight how good it would be to branch on that particular branching object. The results of the presolved children are stored in an object of this type as well as the instructions what to do with each child in case this object is selected for branching.

Definition at line 321 of file [BCP\\_lp\\_branch.hpp](#).

### 4.35.2 Constructor & Destructor Documentation

4.35.2.1 [BCP\\_presolved\\_lp\\_brobj::BCP\\_presolved\\_lp\\_brobj](#) ( [BCP\\_lp\\_branching\\_object](#) \* *candidate* ) [[inline](#)],  
[[explicit](#)]

The only one way to construct a presolved branching object is to create it from a regular branching object.

Definition at line 358 of file [BCP\\_lp\\_branch.hpp](#).

4.35.2.2 [BCP\\_presolved\\_lp\\_brobj::~~BCP\\_presolved\\_lp\\_brobj](#) ( ) [[inline](#)]

The destructor simply deletes every member (deletes every lpres in the vector).

Definition at line 373 of file [BCP\\_lp\\_branch.hpp](#).

### 4.35.3 Member Function Documentation

4.35.3.1 [BCP\\_lp\\_branching\\_object\\*](#) [BCP\\_presolved\\_lp\\_brobj::candidate](#) ( ) [[inline](#)]

Return a pointer to the candidate.

THINK\* : is it necessary to have a non-const version???

Definition at line 388 of file [BCP\\_lp\\_branch.hpp](#).

4.35.3.2 [const BCP\\_lp\\_branching\\_object\\*](#) [BCP\\_presolved\\_lp\\_brobj::candidate](#) ( ) [const](#) [[inline](#)]

Return a const pointer to the candidate.

Definition at line 392 of file [BCP\\_lp\\_branch.hpp](#).

**4.35.3.3** `const BCP_lp_result& BCP_presolved_lp_brobj::lpres ( const int child_ind ) const` `[inline]`

Return a const reference to the presolved results of the *child\_ind*-th child.

Definition at line 397 of file BCP\_lp\_branch.hpp.

**4.35.3.4** `BCP_vec<BCP_child_action>& BCP_presolved_lp_brobj::action ( )` `[inline]`

Return a reference to the actions to be taken.

A non-const method is needed, since this is the easiest way to set the entries. Maybe it'd be cleaner to have a separate set method...

Definition at line 403 of file BCP\_lp\_branch.hpp.

**4.35.3.5** `const BCP_vec<BCP_child_action>& BCP_presolved_lp_brobj::action ( ) const` `[inline]`

Return a const reference to the actions to be taken.

Definition at line 407 of file BCP\_lp\_branch.hpp.

**4.35.3.6** `BCP_vec<BCP_user_data*>& BCP_presolved_lp_brobj::user_data ( )` `[inline]`

Return a reference to the user data vector.

A non-const method is needed, since this is the easiest way to set the entries. Maybe it'd be cleaner to have a separate set method...

Definition at line 414 of file BCP\_lp\_branch.hpp.

**4.35.3.7** `const BCP_vec<BCP_user_data*>& BCP_presolved_lp_brobj::user_data ( ) const` `[inline]`

Return a const reference to the user data vector.

Definition at line 418 of file BCP\_lp\_branch.hpp.

**4.35.3.8** `bool BCP_presolved_lp_brobj::fathomable ( const double objval_limit ) const`

Return true if every children can be fathomed.

(The lower bound for each is above *objval\_limit*.)

**4.35.3.9** `bool BCP_presolved_lp_brobj::had_numerical_problems ( ) const`

Return true if at least one child had numerical difficulties while presolving.

**4.35.3.10** `void BCP_presolved_lp_brobj::get_lower_bounds ( BCP_vec< double > & obj )` `[inline]`

Fill up *obj* with the lower bound on each child.

Definition at line 450 of file BCP\_lp\_branch.hpp.

4.35.3.11 `void BCP_presolved_lp_brobj::set_lower_bounds ( const BCP_vec< double > & obj ) [inline]`

Fill up the lower bounds on the children with the content of `obj`.

Definition at line 459 of file `BCP_lp_branch.hpp`.

4.35.3.12 `void BCP_presolved_lp_brobj::get_results ( OsiSolverInterface & lp, const int child_ind ) [inline]`

Extract the lp results from the LP solver for the `child_ind`-th child.

This is done immediately after presolving the child.

Definition at line 467 of file `BCP_lp_branch.hpp`.

4.35.3.13 `void BCP_presolved_lp_brobj::fake_objective_values ( const double itlim_objval )`

Examine the termination codes for the children and for those that do not have a valid lower bound fake the objective value depending on the termination code:

- primal infeasibility / dual objective limit: `BCP_DBL_MAX`
- iteration limit : maximum of the lower bound at termination and `itlim_objval`
- abandoned : `itlim_objval`

4.35.3.14 `void BCP_presolved_lp_brobj::set_objective_values ( const BCP_vec< double > & obj, const BCP_vec< int > & termcode, const double itlim_objval )`

Set the appropriate fields of all `_lpres` to the given `termcode` and `objval` if the `termcode` is "normal".

If not normal (like the cases in [fake\\_objective\\_values\(\)](#)), then apply the same rules.

The documentation for this class was generated from the following file:

- `BCP_lp_branch.hpp`

## 4.36 BCP\_problem\_core Class Reference

This class describes the core of the MIP problem, the variables/cuts in it as well as the matrix corresponding the core variables and cuts.

```
#include <BCP_problem_core.hpp>
```

Collaboration diagram for `BCP_problem_core`:

### Public Member Functions

#### Constructors and destructor

- [BCP\\_problem\\_core](#) ()  
*The default constructor creates an empty core description: no variables/cuts and an empty matrix.*
- [BCP\\_problem\\_core](#) ([BCP\\_vec](#)< [BCP\\_var\\_core](#) \* > &v, [BCP\\_vec](#)< [BCP\\_cut\\_core](#) \* > &c, [BCP\\_lp\\_relax](#) \*&m)

*This constructor "takes over" the arguments.*

- `~BCP_problem_core ()`

*The desctructor deletes all data members.*

### Query methods

- `size_t varnum () const`

*Return the number of variables in the core.*

- `size_t cutnum () const`

*Return the number of cuts in the core.*

### Packing and unpacking methods

- `void pack (BCP_buffer &buf) const`

*Pack the contents of the core description into the buffer.*

- `void unpack (BCP_buffer &buf)`

*Unpack the contents of the core description from the buffer.*

## Public Attributes

### Data members

- `BCP_vec< BCP_var_core * > vars`

*A vector of pointers to the variables in the core of the problem.*

- `BCP_vec< BCP_cut_core * > cuts`

*A vector of pointers to the cuts in the core of the problem.*

- `BCP_lp_relax * matrix`

*A pointer to the constraint matrix corresponding to the core variables and cuts.*

### 4.36.1 Detailed Description

This class describes the core of the MIP problem, the variables/cuts in it as well as the matrix corresponding the core variables and cuts.

Core cuts and variables never leave the formulation.

Definition at line 31 of file `BCP_problem_core.hpp`.

### 4.36.2 Constructor & Destructor Documentation

#### 4.36.2.1 `BCP_problem_core::BCP_problem_core ( )`

The default constructor creates an empty core description: no variables/cuts and an empty matrix.

#### 4.36.2.2 `BCP_problem_core::BCP_problem_core ( BCP_vec< BCP_var_core * > & v, BCP_vec< BCP_cut_core * > & c, BCP_lp_relax * & m ) [inline]`

This constructor "takes over" the arguments.

The created core description will have the content of the arguments in its data members while the arguments lose their content.

Definition at line 65 of file `BCP_problem_core.hpp`.

#### 4.36.2.3 BCP\_problem\_core::~~BCP\_problem\_core ( )

The destructor deletes all data members.

### 4.36.3 Member Function Documentation

#### 4.36.3.1 size\_t BCP\_problem\_core::varnum ( ) const [inline]

Return the number of variables in the core.

Definition at line 78 of file BCP\_problem\_core.hpp.

#### 4.36.3.2 size\_t BCP\_problem\_core::cutnum ( ) const [inline]

Return the number of cuts in the core.

Definition at line 80 of file BCP\_problem\_core.hpp.

#### 4.36.3.3 void BCP\_problem\_core::pack ( BCP\_buffer & buf ) const

Pack the contents of the core description into the buffer.

#### 4.36.3.4 void BCP\_problem\_core::unpack ( BCP\_buffer & buf )

Unpack the contents of the core description from the buffer.

### 4.36.4 Member Data Documentation

#### 4.36.4.1 BCP\_vec<BCP\_var\_core\*> BCP\_problem\_core::vars

A vector of pointers to the variables in the core of the problem.

These are the variables that always stay in the problem formulation.

Definition at line 48 of file BCP\_problem\_core.hpp.

#### 4.36.4.2 BCP\_vec<BCP\_cut\_core\*> BCP\_problem\_core::cuts

A vector of pointers to the cuts in the core of the problem.

These are the cuts that always stay in the problem formulation.

Definition at line 51 of file BCP\_problem\_core.hpp.

#### 4.36.4.3 BCP\_lp\_relax\* BCP\_problem\_core::matrix

A pointer to the constraint matrix corresponding to the core variables and cuts.

Definition at line 54 of file BCP\_problem\_core.hpp.

The documentation for this class was generated from the following file:

- BCP\_problem\_core.hpp

## 4.37 BCP\_problem\_core\_change Class Reference

This class describes changes in the core of the problem.

```
#include <BCP_problem_core.hpp>
```

Collaboration diagram for BCP\_problem\_core\_change:

### Public Member Functions

#### Constructors and destructor

- [BCP\\_problem\\_core\\_change](#) (BCP\_storage\_t store=BCP\_Storage\_WrtCore)  
*This constructor creates a core change with the given storage.*
- [BCP\\_problem\\_core\\_change](#) (int bvarnum, [BCP\\_var\\_set](#) &vars, int bcutnum, [BCP\\_cut\\_set](#) &cuts)  
*This constructor creates an Explicit core change description.*
- [BCP\\_problem\\_core\\_change](#) (BCP\_storage\_t storage, [BCP\\_problem\\_core\\_change](#) &ocore, [BCP\\_problem\\_core\\_change](#) &ncore)  
*Create a core change describing the changes from `old_bc` to `new_bc`.*
- [~BCP\\_problem\\_core\\_change](#) ()  
*The destructor deletes all data members.*

#### Query methods

- BCP\_storage\_t [storage](#) () const  
*Return the storage type.*
- size\_t [varnum](#) () const  
*Return the number of changed variables (the length of the array `var_ch`).*
- size\_t [cutnum](#) () const  
*Return the number of changed cuts (the length of the array `cut_ch`).*

#### Modifying methods

- [BCP\\_problem\\_core\\_change](#) & [operator=](#) (const [BCP\\_problem\\_core](#) &core)  
*Set the core change description to be an explicit description (in the form of a change) of the given `core`.*
- void [ensure\\_explicit](#) (const [BCP\\_problem\\_core\\_change](#) &expl\_core)  
*If the current storage is not already explicit then replace it with an explicit description of the core generated by applying the currently stored changes to `expl_core` (which of course, must be explicitly stored).*
- void [make\\_wrtcore\\_if\\_shorter](#) (const [BCP\\_problem\\_core\\_change](#) &orig\_core)  
*Replace the current explicitly stored core change with one stored with respect to the explicitly stored original core change in `orig_core` if the WrtCore description is shorter (requires less space to pack into a buffer).*
- void [swap](#) ([BCP\\_problem\\_core\\_change](#) &other)  
*Swap the contents of the current core change with that of `other`.*
- void [update](#) (const [BCP\\_problem\\_core\\_change](#) &expl\_core, const [BCP\\_problem\\_core\\_change](#) &core\_change)  
*Update the current change according to `core_change`.*

#### Packing and unpacking

- int [pack\\_size](#) () const  
*Return the buffer size needed to pack the data in the core change.*
- void [pack](#) ([BCP\\_buffer](#) &buf) const  
*Pack the core change into the buffer.*
- void [unpack](#) ([BCP\\_buffer](#) &buf)  
*Unpack the core change data from the buffer.*

## Public Attributes

### Data members

- [BCP\\_storage\\_t \\_storage](#)  
*Describes how the change is stored.*
- [BCP\\_vec< int > var\\_pos](#)  
*The positions of the core variables (in the `vars` member of [\[BCP\\_problem\\_core\]](#)[\[BCP\\_problem\\_core.html\]](#)) whose bounds and/or stati have changed.*
- [BCP\\_vec< BCP\\_obj\\_change > var\\_ch](#)  
*The new lb/ub/status triplet for each variable for which any of those three have changed.*
- [BCP\\_vec< int > cut\\_pos](#)  
*The positions of the core cuts (in the `cuts` member of [\[BCP\\_problem\\_core\]](#)[\[BCP\\_problem\\_core.html\]](#)) whose bounds and/or stati have changed.*
- [BCP\\_vec< BCP\\_obj\\_change > cut\\_ch](#)  
*The new lb/ub/status triplet for each cut for which any of those three have changed.*

### 4.37.1 Detailed Description

This class describes changes in the core of the problem.

While the set of core variables and cuts always remains the same, their bounds and stati may change during processing. An object of this type may store the description of the bounds and stati of the core of three possible ways:

- explicitly: the bounds and stati of everything in the core is stored in this class. In this case `..._pos` is empty and `..._ch` stores the bounds and stati.
- with respect to parent: this is the case when a sequence of core changes have to be applied to get the current state. Each core change in the sequence describes the changes to be done after the previous change has been applied.
- with respect to core: the current state of the core is given as one set of changes with respect to the very original state of the core.

This class is internal to the framework, the user shouldn't worry about it.

Definition at line 116 of file `BCP_problem_core.hpp`.

### 4.37.2 Constructor & Destructor Documentation

**4.37.2.1** `BCP_problem_core_change::BCP_problem_core_change ( BCP_storage_t store = BCP_Storage_WrtCore )`  
[inline]

This constructor creates a core change with the given storage.

The other members are empty, i.e., the created object contains NoData; is Explicit and empty; is WrtCore or WrtParent but there's no change.

Note that the argument defaults to WrtCore, this constructor is the default constructor as well, and as the default constructor it creates a "no change WrtCore" core change.

Definition at line 169 of file `BCP_problem_core.hpp`.

**4.37.2.2** `BCP_problem_core_change::BCP_problem_core_change ( int bvarnum, BCP_var_set & vars, int bcutnum, BCP_cut_set & cuts )`

This constructor creates an Explicit core change description.

The first `bvarnum` variables in `vars` are the core variables. Similarly for the cuts.

**4.37.2.3** `BCP_problem_core_change::BCP_problem_core_change ( BCP_storage_t storage, BCP_problem_core_change & ocore, BCP_problem_core_change & ncore )`

Create a core change describing the changes from `old_bc` to `new_bc`.

Both core change arguments must have explicit storage. The only reason for passing `storage` as an argument (and not setting it automatically to `WrtParent`) is that `old_bc` might be the original core in which case storage must be set to `WrtCore`.

**4.37.2.4** `BCP_problem_core_change::~BCP_problem_core_change ( ) [inline]`

The destructor deletes all data members.

Definition at line 189 of file `BCP_problem_core.hpp`.

### 4.37.3 Member Function Documentation

**4.37.3.1** `BCP_storage_t BCP_problem_core_change::storage ( ) const [inline]`

Return the storage type.

Definition at line 198 of file `BCP_problem_core.hpp`.

**4.37.3.2** `size_t BCP_problem_core_change::varnum ( ) const [inline]`

Return the number of changed variables (the length of the array `var_ch`).

Definition at line 201 of file `BCP_problem_core.hpp`.

**4.37.3.3** `size_t BCP_problem_core_change::cutnum ( ) const [inline]`

Return the number of changed cuts (the length of the array `cut_ch`).

Definition at line 204 of file `BCP_problem_core.hpp`.

**4.37.3.4** `BCP_problem_core_change& BCP_problem_core_change::operator= ( const BCP_problem_core & core )`

Set the core change description to be an explicit description (in the form of a change) of the given `core`.

**4.37.3.5** `void BCP_problem_core_change::ensure_explicit ( const BCP_problem_core_change & expl_core )`

If the current storage is not already explicit then replace it with an explicit description of the core generated by applying the currently stored changes to `expl_core` (which of course, must be explicitly stored).



NOTE: An exception is thrown if the currently stored change is not stored as explicit or WrtCore; or the storage of `expl_core` is not explicit.

**4.37.3.6** `void BCP_problem_core_change::make_wrtcore_if_shorter ( const BCP_problem_core_change & orig_core )`

Replace the current explicitly stored core change with one stored with respect to the explicitly stored original core change in `orig_core` if the WrtCore description is shorter (requires less space to pack into a buffer).

NOTE: An exception is thrown if either the current storage or that of `expl_core` is not explicit.

**4.37.3.7** `void BCP_problem_core_change::swap ( BCP_problem_core_change & other )`

Swap the contents of the current core change with that of `other`.

**4.37.3.8** `void BCP_problem_core_change::update ( const BCP_problem_core_change & expl_core, const BCP_problem_core_change & core_change )`

Update the current change according to `core_change`.

If the storage type of `core_change` is

- NoData or Explicit then it is simply copied over into this change.
- WrtParent then the current change is supposed to be the parent and this explicitly stored core change will be updated with the changes in `core_change` (an exception is thrown if the current change is not explicitly stored).
- WrtCore storage then the current change will be replaced by `expl_core` updated with `core_change` (the storage of `expl_core` must be explicit or an exception is thrown).

NOTE: When this function exits the stored change will have either explicit or NoData storage.

**4.37.3.9** `int BCP_problem_core_change::pack_size ( ) const`

Return the buffer size needed to pack the data in the core change.

**4.37.3.10** `void BCP_problem_core_change::pack ( BCP_buffer & buf ) const`

Pack the core change into the buffer.

**4.37.3.11** `void BCP_problem_core_change::unpack ( BCP_buffer & buf )`

Unpack the core change data from the buffer.

## 4.37.4 Member Data Documentation

**4.37.4.1** `BCP_storage_t BCP_problem_core_change::_storage`

Describes how the change is stored.

The interpretation of the other data members depends on the storage type.

- `BCP_Storage_NoData`: when no data is stored (i.e., the change is not described yet) the other members are undefined.
- `BCP_Storage_WrtCore`: with respect to core storage (as explained in the class description).
- `BCP_Storage_WrtParent`: with respect to parent storage (as explained in the class description).
- `BCP_Storage_Explicit`: Explicit storage (as explained in the class description).

Definition at line 143 of file `BCP_problem_core.hpp`.

#### 4.37.4.2 `BCP_vec<int> BCP_problem_core_change::var_pos`

The positions of the core variables (in the `vars` member of `[BCP_problem_core]`{`BCP_problem_core.html`}) whose bounds and/or stati have changed.

Definition at line 147 of file `BCP_problem_core.hpp`.

#### 4.37.4.3 `BCP_vec<BCP_obj_change> BCP_problem_core_change::var_ch`

The new lb/ub/status triplet for each variable for which any of those three have changed.

Definition at line 150 of file `BCP_problem_core.hpp`.

#### 4.37.4.4 `BCP_vec<int> BCP_problem_core_change::cut_pos`

The positions of the core cuts (in the `cuts` member of `[BCP_problem_core]`{`BCP_problem_core.html`}) whose bounds and/or stati have changed.

Definition at line 154 of file `BCP_problem_core.hpp`.

#### 4.37.4.5 `BCP_vec<BCP_obj_change> BCP_problem_core_change::cut_ch`

The new lb/ub/status triplet for each cut for which any of those three have changed.

Definition at line 157 of file `BCP_problem_core.hpp`.

The documentation for this class was generated from the following file:

- `BCP_problem_core.hpp`

## 4.38 `BCP_process` Class Reference

Inheritance diagram for `BCP_process`:

### 4.38.1 Detailed Description

Definition at line 16 of file `BCP_process.hpp`.

The documentation for this class was generated from the following file:

- `BCP_process.hpp`

## 4.39 BCP\_row Class Reference

This class holds a row in a compressed form.

```
#include <BCP_matrix.hpp>
```

Inheritance diagram for BCP\_row:

Collaboration diagram for BCP\_row:

### Public Member Functions

#### Query methods

- double [LowerBound](#) () const  
*Return the lower bound.*
- double [UpperBound](#) () const  
*Return the upper bound.*

#### General modifying methods

- void [LowerBound](#) (double lb)  
*Set the lower bound to the given value.*
- void [UpperBound](#) (double ub)  
*Set the upper bound to the given value.*
- [BCP\\_row](#) & [operator=](#) (const [BCP\\_row](#) &x)  
*Assignment operator: copy over the contents of x.*
- void [assign](#) (const int size, int \*&ElementIndices, double \*&ElementValues, const double LB, const double UB)  
*Set the lower and upper bounds to the given values.*
- void [copy](#) (const int size, const int \*ElementIndices, const double \*ElementValues, const double LB, const double UB)  
*Copy the arguments into the appropriate data members.*
- void [copy](#) ([BCP\\_vec](#)< int >::const\_iterator firstind, [BCP\\_vec](#)< int >::const\_iterator lastind, [BCP\\_vec](#)< double >::const\_iterator firstval, [BCP\\_vec](#)< double >::const\_iterator lastval, const double LB, const double UB)  
*Same as the other [copy\(\)](#) method, except that instead of using vectors the indices (values) are given in [firstind, lastind) ([firstval, lastval]).*

#### Constructors / Destructor

- [BCP\\_row](#) ()  
*The default constructor creates an empty row with -infinity as lower and +infinity as upper bound.*
- [BCP\\_row](#) (const [BCP\\_row](#) &x)  
*The copy constructor makes a copy of x.*
- [BCP\\_row](#) ([BCP\\_vec](#)< int >::const\_iterator firstind, [BCP\\_vec](#)< int >::const\_iterator lastind, [BCP\\_vec](#)< double >::const\_iterator firstval, [BCP\\_vec](#)< double >::const\_iterator lastval, const double LB, const double UB)  
*This constructor acts exactly like the [copy](#) method with the same argument list.*
- [BCP\\_row](#) (const int size, int \*&ElementIndices, double \*&ElementValues, const double LB, const double UB)  
*This constructor acts exactly like the [assign](#) method with the same argument list.*
- [BCP\\_row](#) (const [CoinPackedVectorBase](#) &vec, const double LB, const double UB)
- [~BCP\\_row](#) ()  
*The destructor deletes all data members.*

## Protected Attributes

### Data members

- double [\\_LowerBound](#)  
*The lower bound corresponding to the row.*
- double [\\_UpperBound](#)  
*The upper bound corresponding to the row.*

### 4.39.1 Detailed Description

This class holds a row in a compressed form.

That is, it is a packed vector with a lower and upper bound.

Definition at line 152 of file BCP\_matrix.hpp.

### 4.39.2 Constructor & Destructor Documentation

#### 4.39.2.1 `BCP_row::BCP_row( )` `[inline]`

The default constructor creates an empty row with -infinity as lower and +infinity as upper bound.

Definition at line 228 of file BCP\_matrix.hpp.

#### 4.39.2.2 `BCP_row::BCP_row( const BCP_row & x )` `[inline]`

The copy constructor makes a copy of `x`.

Definition at line 231 of file BCP\_matrix.hpp.

#### 4.39.2.3 `BCP_row::BCP_row( BCP_vec< int >::const_iterator firstind, BCP_vec< int >::const_iterator lastind, BCP_vec< double >::const_iterator firstval, BCP_vec< double >::const_iterator lastval, const double LB, const double UB )` `[inline]`

This constructor acts exactly like the `copy` method with the same argument list.

Definition at line 236 of file BCP\_matrix.hpp.

#### 4.39.2.4 `BCP_row::BCP_row( const int size, int *& ElementIndices, double *& ElementValues, const double LB, const double UB )` `[inline]`

This constructor acts exactly like the `assign` method with the same argument list.

Definition at line 246 of file BCP\_matrix.hpp.

#### 4.39.2.5 `BCP_row::~BCP_row( )` `[inline]`

The destructor deletes all data members.

Definition at line 257 of file BCP\_matrix.hpp.

### 4.39.3 Member Function Documentation

#### 4.39.3.1 `double BCP_row::LowerBound ( ) const` `[inline]`

Return the lower bound.

Definition at line 167 of file BCP\_matrix.hpp.

#### 4.39.3.2 `double BCP_row::UpperBound ( ) const` `[inline]`

Return the upper bound.

Definition at line 169 of file BCP\_matrix.hpp.

#### 4.39.3.3 `void BCP_row::LowerBound ( double lb )` `[inline]`

Set the lower bound to the given value.

Definition at line 176 of file BCP\_matrix.hpp.

#### 4.39.3.4 `void BCP_row::UpperBound ( double ub )` `[inline]`

Set the upper bound to the given value.

Definition at line 178 of file BCP\_matrix.hpp.

#### 4.39.3.5 `BCP_row& BCP_row::operator= ( const BCP_row & x )` `[inline]`

Assignment operator: copy over the contents of *x*.

Definition at line 181 of file BCP\_matrix.hpp.

#### 4.39.3.6 `void BCP_row::assign ( const int size, int * ElementIndices, double * ElementValues, const double LB, const double UB )` `[inline]`

Set the lower and upper bounds to the given values.

Also invokes the assign method of the underlying packed vector.

Definition at line 191 of file BCP\_matrix.hpp.

#### 4.39.3.7 `void BCP_row::copy ( const int size, const int * ElementIndices, const double * ElementValues, const double LB, const double UB )` `[inline]`

Copy the arguments into the appropriate data members.

Definition at line 200 of file BCP\_matrix.hpp.

```
4.39.3.8 void BCP_row::copy ( BCP_vec< int >::const_iterator firstind, BCP_vec< int >::const_iterator lastind, BCP_vec<
double >::const_iterator firstval, BCP_vec< double >::const_iterator lastval, const double LB, const double UB )
[inline]
```

Same as the other `copy()` method, except that instead of using vectors the indices (values) are given in `[firstind, lastind)` (`[firstval, lastval)`).

Definition at line 211 of file `BCP_matrix.hpp`.

#### 4.39.4 Member Data Documentation

```
4.39.4.1 double BCP_row::_LowerBound [protected]
```

The lower bound corresponding to the row.

Definition at line 157 of file `BCP_matrix.hpp`.

```
4.39.4.2 double BCP_row::_UpperBound [protected]
```

The upper bound corresponding to the row.

Definition at line 159 of file `BCP_matrix.hpp`.

The documentation for this class was generated from the following file:

- `BCP_matrix.hpp`

## 4.40 BCP\_scheduler Class Reference

### Public Member Functions

- [BCP\\_scheduler\(\)](#)  
*Default constructor.*
- void [setParams](#) (double OverEstimationStatic, double SwitchToRateThreshold, double TimeRootNodeSolve, double FactorTimeHorizon, double OverEstimationRate, double MaxNodeIdRatio, int MaxNodeIdNum, int MaxSbIdNum, int MinSbIdNum)  
*Method for setting scheduler parameters.*
- void [add\\_free\\_ids](#) (int numIds, const int \*ids)  
*Pass in a list of freelds\_ to add.*
- int [request\\_sb\\_ids](#) (int numIds, int \*ids)  
*Request for a number of id's to do some strong branching.*
- void [release\\_sb\\_id](#) (int id)  
*Gives back to scheduler an id used for strong branching.*
- int [request\\_node\\_id](#) ()  
*Request an id for processing nodes.*
- void [release\\_node\\_id](#) (int id)  
*Give back an id to scheduler used for processing a node.*
- bool [has\\_free\\_node\\_id](#) () const  
*Decide if there is an id that can be returned for processin a node.*
- int [numNodeIds](#) () const

*Return the number of busy LP processes.*

- int `maxNodeIds` () const

*Return the maximum possible number of busy LP processes.*

- double `node_idle` (int p)

*Return how much time did process p spent idling as a node process.*

- double `sb_idle` (int p)

*Return how much time did process p spent idling as a SB process.*

- void `update_idle_times` ()

*Update idle times with the last idle time.*

#### 4.40.1 Detailed Description

Definition at line 49 of file BCP\_process.hpp.

#### 4.40.2 Constructor & Destructor Documentation

##### 4.40.2.1 BCP\_scheduler::BCP\_scheduler ( )

Default constructor.

#### 4.40.3 Member Function Documentation

##### 4.40.3.1 void BCP\_scheduler::setParams ( double *OverEstimationStatic*, double *SwitchToRateThreshold*, double *TimeRootNodeSolve*, double *FactorTimeHorizon*, double *OverEstimationRate*, double *MaxNodeIdRatio*, int *MaxNodeIdNum*, int *MaxSbIdNum*, int *MinSbIdNum* )

Method for setting scheduler parameters.

Parameters

<i>OverEstimationStatic</i>	Factor for providing more IDs in static strategy.
<i>SwitchToRateThreshold</i>	When more than SwitchToRateThreshold times the number of strong-branching CPUs are busy, which to rate-based strategy.
<i>TimeRootNodeSolve</i>	Time to solve root node NLP (in secs)
<i>FactorTimeHorizon</i>	This number times TimeRootNodeSolve is used to compute the rates
<i>OverEstimationRate</i>	Factor for providing more IDs in rate-based strategy.
<i>MaxNodeIdRatio</i>	At most this fraction of the total number of ids can be used as a nodeid.
<i>MaxNodeIdNum</i>	At most this many ids can be used as a nodeid.

##### 4.40.3.2 void BCP\_scheduler::add\_free\_ids ( int *numIds*, const int \* *ids* )

Pass in a list of freelds\_ to add.

#### 4.40.3.3 `int BCP_scheduler::request_sb_ids ( int numIds, int * ids )`

Request for a number of id's to do some strong branching.

NOTE: ids must be already allocated to be at least `numIds` size.

##### Parameters

<i>numIds</i>	: number of ids requested
<i>ids</i>	: filled vector with the number of ids served.

##### Returns

number of ids served.

#### 4.40.3.4 `void BCP_scheduler::release_sb_id ( int id )`

Gives back to scheduler an id used for strong branching.

#### 4.40.3.5 `int BCP_scheduler::request_node_id ( )`

Request an id for processing nodes.

##### Returns

id number or -1 if none is available.

The documentation for this class was generated from the following file:

- `BCP_process.hpp`

## 4.41 BCP\_single\_environment Class Reference

Inheritance diagram for `BCP_single_environment`:

Collaboration diagram for `BCP_single_environment`:

### Public Member Functions

- `int register_process (USER_initialize *user_init)`  
A process can register (receive its process id) with the message passing environment.
- `int parent_process ()`  
Return the process id of the parent process (the process that spawned the current process).
- `bool alive (const int pid)`  
Test if the process given by the argument is alive or not.
- `const int * alive (int num, const int *pids)`  
Test if the processes given by the process array in the argument are alive or not.
- `void send (const int target, const BCP_message_tag tag)`  
Send an empty message (message tag only) to the process given by the first argument.
- `void send (const int target, const BCP_message_tag tag, const BCP_buffer &buf)`



*Send the message in the buffer with the given message tag to the process given by the first argument.*

- void [multicast](#) (int num, const int \*targets, const BCP\_message\_tag tag)

*Send an empty message (message tag only) to all the processes in the process array.*

- void [multicast](#) (int num, const int \*targets, const BCP\_message\_tag tag, const [BCP\\_buffer](#) &buf)

*Send the message in the buffer with the given message tag to all processes in the process array.*

- void [receive](#) (const int source, const BCP\_message\_tag tag, [BCP\\_buffer](#) &buf, const double timeout)

*Blocking receive with timeout.*

- bool [probe](#) (const int source, const BCP\_message\_tag tag)

*Probe if there are any messages from the given process with the given message tag.*

- int [start\\_process](#) (const [BCP\\_string](#) &exe, const bool debug)

*Spawn a new process.*

- int [start\\_process](#) (const [BCP\\_string](#) &exe, const [BCP\\_string](#) &machine, const bool debug)

*Spawn a new process on the machine specified by the second argument.*

- bool [start\\_processes](#) (const [BCP\\_string](#) &exe, const int proc\_num, const bool debug, int \*ids)

*Spawn proc\_num processes, all with the same executable.*

- bool [start\\_processes](#) (const [BCP\\_string](#) &exe, const int proc\_num, const [BCP\\_vec](#)< [BCP\\_string](#) > &machines, const bool debug, int \*ids)

*Spawn proc\_num processes on the machines given by the third argument, all with the same executable.*

#### 4.41.1 Detailed Description

Definition at line 16 of file BCP\_message\_single.hpp.

#### 4.41.2 Member Function Documentation

4.41.2.1 int BCP\_single\_environment::register\_process ( [USER\\_initialize](#) \* *user\_init* ) [virtual]

A process can register (receive its process id) with the message passing environment.

Implements [BCP\\_message\\_environment](#).

4.41.2.2 int BCP\_single\_environment::parent\_process ( ) [virtual]

Return the process id of the parent process (the process that spawned the current process).

Returns null if the process does not have a parent.

Implements [BCP\\_message\\_environment](#).

4.41.2.3 bool BCP\_single\_environment::alive ( const int *pid* ) [virtual]

Test if the process given by the argument is alive or not.

Return true if alive, false otherwise.

Implements [BCP\\_message\\_environment](#).

4.41.2.4 `const int* BCP_single_environment::alive ( int num, const int * pids )` [virtual]

Test if the processes given by the process array in the argument are alive or not.

Return a pointer to the first dead process, or to the end of the array if there are no dead processes.

Implements [BCP\\_message\\_environment](#).

4.41.2.5 `void BCP_single_environment::send ( const int target, const BCP_message_tag tag )` [virtual]

Send an empty message (message tag only) to the process given by the first argument.

Implements [BCP\\_message\\_environment](#).

4.41.2.6 `void BCP_single_environment::send ( const int target, const BCP_message_tag tag, const BCP_buffer & buf )`  
[virtual]

Send the message in the buffer with the given message tag to the process given by the first argument.

Implements [BCP\\_message\\_environment](#).

4.41.2.7 `void BCP_single_environment::multicast ( int num, const int * targets, const BCP_message_tag tag )` [virtual]

Send an empty message (message tag only) to all the processes in the process array.

Implements [BCP\\_message\\_environment](#).

4.41.2.8 `void BCP_single_environment::multicast ( int num, const int * targets, const BCP_message_tag tag, const BCP_buffer & buf )` [virtual]

Send the message in the buffer with the given message tag to all processes in the process array.

Implements [BCP\\_message\\_environment](#).

4.41.2.9 `void BCP_single_environment::receive ( const int source, const BCP_message_tag tag, BCP_buffer & buf, const double timeout )` [virtual]

Blocking receive with timeout.

Wait until a message is received from the given process with the given message tag within the given timelimit. If `timeout` is positive then the receive routine times out after this many seconds. If it is negative then the method blocks until a message is received. The received message is saved into the buffer. With a 0 pointer as the first argument (or the predefined `BCP_AnyProcess` process id can be used) a message from any process will be accepted. Messages with any message tag are accepted if the second argument is `BCP_Msg_AnyMessage`.

Implements [BCP\\_message\\_environment](#).

4.41.2.10 `bool BCP_single_environment::probe ( const int source, const BCP_message_tag tag )` [virtual]

Probe if there are any messages from the given process with the given message tag.

Return true if such a message is found, false otherwise. Note that the message is not "read", only its existence is checked. Similarly as above, the wild cards `BCP_AnyProcess` and `BCP_Msg_AnyMessage` can be used.

Implements [BCP\\_message\\_environment](#).

4.41.2.11 `int BCP_single_environment::start_process ( const BCP_string & exe, const bool debug ) [virtual]`

Spawn a new process.

The first argument contains the path to the executable to be spawned. If the second argument is set to true, then the executable is spawned under a debugger.

Implements [BCP\\_message\\_environment](#).

4.41.2.12 `int BCP_single_environment::start_process ( const BCP_string & exe, const BCP_string & machine, const bool debug ) [virtual]`

Spawn a new process on the machine specified by the second argument.

Implements [BCP\\_message\\_environment](#).

4.41.2.13 `bool BCP_single_environment::start_processes ( const BCP_string & exe, const int proc_num, const bool debug, int * ids ) [virtual]`

Spawn `proc_num` processes, all with the same executable.

NOTE: `ids` must be allocated already and have enough space for `proc_num` entries.

Returns

true/false depending on success.

Implements [BCP\\_message\\_environment](#).

4.41.2.14 `bool BCP_single_environment::start_processes ( const BCP_string & exe, const int proc_num, const BCP_vec< BCP_string > & machines, const bool debug, int * ids ) [virtual]`

Spawn `proc_num` processes on the machines given by the third argument, all with the same executable.

NOTE: `ids` must be allocated already and have enough space for `proc_num` entries.

Returns

true/false depending on success.

Implements [BCP\\_message\\_environment](#).

The documentation for this class was generated from the following file:

- `BCP_message_single.hpp`

## 4.42 BCP\_slave\_params Struct Reference

NO OLD DOC.

```
#include <BCP_tm.hpp>
```

Collaboration diagram for BCP\_slave\_params:

#### 4.42.1 Detailed Description

NO OLD DOC.

Definition at line 57 of file BCP\_tm.hpp.

The documentation for this struct was generated from the following file:

- BCP\_tm.hpp

### 4.43 BCP\_solution Class Reference

This is the abstract base class for a solution to a Mixed Integer Programming problem.

```
#include <BCP_solution.hpp>
```

Inheritance diagram for BCP\_solution:

#### Public Member Functions

- virtual [~BCP\\_solution](#) ()  
*The virtual destructor ensures that the correct destructor is invoked.*
- virtual double [objective\\_value](#) () const =0  
*The method returning the objective value of the solution.*

#### 4.43.1 Detailed Description

This is the abstract base class for a solution to a Mixed Integer Programming problem.

Definition at line 14 of file BCP\_solution.hpp.

#### 4.43.2 Constructor & Destructor Documentation

4.43.2.1 `virtual BCP_solution::~~BCP_solution ( ) [inline],[virtual]`

The virtual destructor ensures that the correct destructor is invoked.

Definition at line 17 of file BCP\_solution.hpp.

#### 4.43.3 Member Function Documentation

4.43.3.1 `virtual double BCP_solution::objective_value ( ) const [pure virtual]`

The method returning the objective value of the solution.

Implemented in [BCP\\_solution\\_generic](#).

The documentation for this class was generated from the following file:

- BCP\_solution.hpp

## 4.44 BCP\_solution\_generic Class Reference

This class holds a MIP feasible primal solution.

```
#include <BCP_solution.hpp>
```

Inheritance diagram for BCP\_solution\_generic:

Collaboration diagram for BCP\_solution\_generic:

### Public Member Functions

- [BCP\\_solution\\_generic](#) (bool delvars=true)  
*The default constructor creates a solution with zero objective value.*
- virtual [~BCP\\_solution\\_generic](#) ()  
*The destructor deletes the data members.*
- virtual double [objective\\_value](#) () const  
*Return the objective value of the solution.*
- void [set\\_objective\\_value](#) (double v)  
*Set the objective value of the solution.*
- void [display](#) () const  
*Display the solution.*
- void [add\\_entry](#) (BCP\_var \*var, double value)  
*Append a variable and the corresponding value to the end of the appropriate vectors.*

### Public Attributes

#### Data members

- double [\\_objective](#)  
*The objective value of the solution.*
- bool [\\_delete\\_vars](#)  
*An indicator to show whether the pointers in `_vars` should be deleted upon destruction or not.*
- [BCP\\_vec< BCP\\_var \\* > \\_vars](#)  
*Vector of variables that are at nonzero level in the solution.*
- [BCP\\_vec< double > \\_values](#)  
*Values of these variables in the solution.*

#### 4.44.1 Detailed Description

This class holds a MIP feasible primal solution.

The default [BCP\\_lp\\_user::pack\\_feasible\\_solution\(\)](#) uses this class to pack an MIP feasible solution, but the user can use this class (instead of using [BCP\\_solution](#) as a base class to derive a different solution holder. (This might be necessary if, for example, the dual values are of importance as well.) The virtual destructor and virtual member functions make this easy.

Definition at line 33 of file [BCP\\_solution.hpp](#).

### 4.44.2 Constructor & Destructor Documentation

#### 4.44.2.1 `BCP_solution_generic::BCP_solution_generic ( bool delvars = true )` `[inline]`

The default constructor creates a solution with zero objective value.

Definition at line 50 of file `BCP_solution.hpp`.

#### 4.44.2.2 `virtual BCP_solution_generic::~~BCP_solution_generic ( )` `[inline]`, `[virtual]`

The destructor deletes the data members.

Note that it purges `_vars` (i.e., deletes the variables the pointers in `_vars` point to) only if the `_delete_vars` member is true.

Definition at line 55 of file `BCP_solution.hpp`.

### 4.44.3 Member Function Documentation

#### 4.44.3.1 `virtual double BCP_solution_generic::objective_value ( ) const` `[inline]`, `[virtual]`

Return the objective value of the solution.

Implements [BCP\\_solution](#).

Definition at line 61 of file `BCP_solution.hpp`.

#### 4.44.3.2 `void BCP_solution_generic::set_objective_value ( double v )` `[inline]`

Set the objective value of the solution.

Definition at line 64 of file `BCP_solution.hpp`.

#### 4.44.3.3 `void BCP_solution_generic::display ( ) const`

Display the solution.

#### 4.44.3.4 `void BCP_solution_generic::add_entry ( BCP_var * var, double value )` `[inline]`

Append a variable and the corresponding value to the end of the appropriate vectors.

This method is used when unpacking the solution.

Definition at line 71 of file `BCP_solution.hpp`.

### 4.44.4 Member Data Documentation

#### 4.44.4.1 `double BCP_solution_generic::_objective`

The objective value of the solution.

Definition at line 38 of file `BCP_solution.hpp`.

#### 4.44.4.2 `bool BCP_solution_generic::_delete_vars`

An indicator to show whether the pointers in `_vars` should be deleted upon destruction or not.

By default they are not.

Definition at line 41 of file `BCP_solution.hpp`.

#### 4.44.4.3 `BCP_vec<BCP_var*> BCP_solution_generic::_vars`

Vector of variables that are at nonzero level in the solution.

Definition at line 43 of file `BCP_solution.hpp`.

#### 4.44.4.4 `BCP_vec<double> BCP_solution_generic::_values`

Values of these variables in the solution.

Definition at line 45 of file `BCP_solution.hpp`.

The documentation for this class was generated from the following file:

- `BCP_solution.hpp`

## 4.45 BCP\_string Class Reference

This class is a very simple implementation of a constant length string.

```
#include <BCP_string.hpp>
```

### 4.45.1 Detailed Description

This class is a very simple implementation of a constant length string.

Using it one can avoid some memory errors related to using functions operating on C style strings.

Definition at line 13 of file `BCP_string.hpp`.

The documentation for this class was generated from the following file:

- `BCP_string.hpp`

## 4.46 BCP\_tm\_flags Struct Reference

NO OLD DOC.

```
#include <BCP_tm.hpp>
```

### Public Attributes

- `bool root_pricing_unpacked`

*Set to true if the result of root pricing is already unpacked.*

### 4.46.1 Detailed Description

NO OLD DOC.

Definition at line 74 of file BCP\_tm.hpp.

### 4.46.2 Member Data Documentation

#### 4.46.2.1 bool BCP\_tm\_flags::root\_pricing\_unpacked

Set to true if the result of root pricing is already unpacked.

Important in a single process environment, so we don't unpack things twice.

Definition at line 78 of file BCP\_tm.hpp.

The documentation for this struct was generated from the following file:

- BCP\_tm.hpp

## 4.47 BCP\_tm\_node Class Reference

Inheritance diagram for BCP\_tm\_node:

Collaboration diagram for BCP\_tm\_node:

### Public Member Functions

#### Constructors and destructor

- **BCP\_tm\_node** (int level, [BCP\\_node\\_change](#) \*desc)
- **~BCP\_tm\_node** ()

#### Query methods

- int **index** () const
- int **child\_num** () const
- int **birth\_index** () const
- [BCP\\_tm\\_node](#) \* **child** (int ind)
- [BCP\\_tm\\_node](#) \* **parent** ()
- const [BCP\\_tm\\_node](#) \* **child** (int ind) const
- const [BCP\\_tm\\_node](#) \* **parent** () const

#### Modifying methods

- int **mark\_descendants\_for\_deletion** ()
- void **remove\_child** ([BCP\\_tm\\_node](#) \*node)
- void **reserve\_child\_num** (int num)
- void **new\_child** ([BCP\\_tm\\_node](#) \*node)



### Data members

- static int **num\_local\_nodes**
- static int **num\_remote\_nodes**
- BCP\_tm\_node\_status **status**
- int **\_index**
- BCP\_tm\_node \* **\_parent**
- int **\_birth\_index**
- BCP\_vec< BCP\_tm\_node \* > **\_children**
- int **lp**
- int **cg**
- int **cp**
- int **vg**
- int **vp**
- int **\_processed\_leaf\_num**
- int **\_pruned\_leaf\_num**
- int **\_tobepriced\_leaf\_num**
- int **\_leaf\_num**
- int **\_core\_storage**:4
- int **\_var\_storage**:4
- int **\_cut\_storage**:4
- int **\_ws\_storage**:4
- int **\_locally\_stored**:2
- int **\_data\_location**:30
- BCP\_tm\_node\_data **\_data**

#### 4.47.1 Detailed Description

Definition at line 60 of file BCP\_tm\_node.hpp.

The documentation for this class was generated from the following file:

- BCP\_tm\_node.hpp

## 4.48 BCP\_tm\_node\_data Class Reference

Collaboration diagram for BCP\_tm\_node\_data:

#### 4.48.1 Detailed Description

Definition at line 51 of file BCP\_tm\_node.hpp.

The documentation for this class was generated from the following file:

- BCP\_tm\_node.hpp

## 4.49 BCP\_tm\_node\_to\_send Class Reference

Collaboration diagram for BCP\_tm\_node\_to\_send:

## Public Member Functions

- bool `send` ()  
*return true or false depending on whether the node was really sent out or it's still waiting for some data*
- bool `receive_node_desc` (BCP\_buffer &buf)  
*return true if has everything to send the thing off to the LP.*
- bool `receive_vars` (BCP\_buffer &buf)  
*return true if has everything to send the thing off to the LP.*
- bool `receive_cuts` (BCP\_buffer &buf)  
*return true if has everything to send the thing off to the LP.*

### 4.49.1 Detailed Description

Definition at line 241 of file BCP\_tm\_node.hpp.

### 4.49.2 Member Function Documentation

#### 4.49.2.1 bool BCP\_tm\_node\_to\_send::receive\_node\_desc ( BCP\_buffer & buf )

return true if has everything to send the thing off to the LP.

Actually, it sends it off, so if this method returns true then then object can be deleted

#### 4.49.2.2 bool BCP\_tm\_node\_to\_send::receive\_vars ( BCP\_buffer & buf )

return true if has everything to send the thing off to the LP.

Actually, it sends it off, so if this method returns true then then object can be deleted

#### 4.49.2.3 bool BCP\_tm\_node\_to\_send::receive\_cuts ( BCP\_buffer & buf )

return true if has everything to send the thing off to the LP.

Actually, it sends it off, so if this method returns true then then object can be deleted

The documentation for this class was generated from the following file:

- BCP\_tm\_node.hpp

## 4.50 BCP\_tm\_par Struct Reference

Parameters used in the Tree Manager process.

```
#include <BCP_tm_param.hpp>
```

Inheritance diagram for BCP\_tm\_par:

## Public Types

- enum `chr_params` {  
`DebugLpProcesses`, `DebugCgProcesses`, `DebugVgProcesses`, `DebugCpProcesses`,  
`DebugVpProcesses`, `GenerateVars`, `MessagePassingIsSerial`, `ReportWhenDefaultIsExecuted`,  
`TrimTreeBeforeNewPhase`, `RemoveExploredBranches`, `VerbosityShutUp`, `TmVerb_First`,  
`TmVerb_AllFeasibleSolutionValue`, `TmVerb_AllFeasibleSolution`, `TmVerb_BetterFeasibleSolutionValue`, `TmVerb_BetterFeasibleSolution`,  
`TmVerb_BestFeasibleSolution`, `TmVerb_NewPhaseStart`, `TmVerb_PrunedNodeInfo`, `TmVerb_TimeOfImprovingSolution`,  
`TmVerb_TrimmedNum`, `TmVerb_FinalStatistics`, `TmVerb_ReportDefault`, `TmVerb_Last` }  
*Character parameters.*
- enum `int_params` {  
`WarmstartInfo`, `MaxHeapSize`, `TmVerb_SingleLineInfoFrequency`, `TreeSearchStrategy`,  
`NiceLevel`, `LpProcessNum`, `CgProcessNum`, `CpProcessNum`,  
`VgProcessNum`, `VpProcessNum`, `TmTimeout`, `LPscheduler_MaxNodeIdNum`,  
`LPscheduler_MaxSbldNum`, `LPscheduler_MinSbldNum` }  
*Integer parameters.*
- enum `dbl_params` {  
`UnconditionalDiveProbability`, `QualityRatioToAllowDiving_HasUB`, `QualityRatioToAllowDiving_NoUB`, `Granularity`,  
`MaxRunTime`, `TerminationGap_Absolute`, `TerminationGap_Relative`, `UpperBound`,  
`LPscheduler_OverEstimationStatic` }  
*Double parameters.*
- enum `str_params` { `SaveRootCutsTo`, `ReadRootCutsFrom`, `ExecutableName`, `LogFileName` }  
*String parameters.*
- enum `str_array_params` {  
`LpMachines`, `CgMachines`, `VgMachines`, `CpMachines`,  
`VpMachines` }  
*???*

### 4.50.1 Detailed Description

Parameters used in the Tree Manager process.

These parameters can be set in the original parameter file by including the following line:

```
BCP_{parameter name} {parameter value}
```

Definition at line 10 of file `BCP_tm_param.hpp`.

### 4.50.2 Member Enumeration Documentation

#### 4.50.2.1 enum `BCP_tm_par::chr_params`

Character parameters.

Most of these variables are used as booleans (true = 1, false = 0).

#### Enumerator

**`DebugLpProcesses`** Indicates whether to debug LP processes or not. Values: 1 (true), 0 (false). Default: 0.

**`DebugCgProcesses`** Indicates whether to debug Cut Generator processes or not. Values: 1 (true), 0 (false). Default: 0.

- DebugVgProcesses** Indicates whether to debug Variable Generator processes or not. Values: 1 (true), 0 (false). Default: 0.
- DebugCpProcesses** Indicates whether to debug Cut Pool processes or not. Values: 1 (true), 0 (false). Default: 0.
- DebugVpProcesses** Indicates whether to debug Variable Pool processes or not. Values: 1 (true), 0 (false). Default: 0.
- GenerateVars** Indicates whether to variable generation will take place or not. Values: 1 (true), 0 (false). Default: 0.
- MessagePassingIsSerial** Indicates whether message passing is serial (all processes are on the same processor) or not. Values: 1 (true), 0 (false). Default: 0.
- ReportWhenDefaultIsExecuted** Print out a message when the default version of an overridable method is executed. Default: 1.
- TrimTreeBeforeNewPhase** Indicates whether to trim the search tree before a new phase. Values: 1 (true), 0 (false). Default: 0.
- RemoveExploredBranches** Indicates whether that part of the tree that's completely explored should be freed as soon as possible. This conserves memory, but may make it harder to track what's happening in the tree. Values: 1 (true), 0 (false). Default: 1.
- VerbosityShutUp** A flag that instructs BCP to be (almost) absolutely silent. It zeros out all the XxVerb flags *even if the verbosity flag is set to 1 later in the parameter file*. Exceptions (flags whose status is not changed) are: TmVerb\_SingleLineInfoFrequency, TmVerb\_FinalStatistics and TmVerb\_BestFeasibleSolution. Default: 0.
- TmVerb\_First** Verbosity flags for the tree manager. Just a marker for the first TmVerb
- TmVerb\_AllFeasibleSolutionValue** Print the value of *any* integer feasible solution found. (BCP\_tm\_prob::process\_message)
- TmVerb\_AllFeasibleSolution** Invoke the user-written "display\_feasible\_solution" function if any\* feasible solution is found. (BCP\_tm\_prob::process\_message)
- TmVerb\_BetterFeasibleSolutionValue** Print the value of the integer solution when a solution better than the current best solution is found. (BCP\_tm\_prob::process\_message)
- TmVerb\_BetterFeasibleSolution** Invoke the user-written "display\_feasible\_solution" function if a better integral feasible solution is found. (BCP\_tm\_prob::process\_message)
- TmVerb\_BestFeasibleSolution** Invoke "display\_feasible\_solution" user routine for the best feasible solution after the entire tree is processed. (BCP\_tm\_wrapup)
- TmVerb\_NewPhaseStart** Print the "Starting phase x" line. (BCP\_tm\_tasks\_before\_new\_phase)
- TmVerb\_PrunedNodeInfo** Print information about nodes that are pruned by bound in the tree manager. These nodes would be returned by the LP if they were sent there, so prune them in the TM instead. (BCP\_tm\_start\_one\_node)
- TmVerb\_TimeOfImprovingSolution** Print the time (and the solution value and solution if the above paramters are set appropriately) when any/better solution is found. (BCP\_tm\_prob::process\_message)
- TmVerb\_TrimmedNum** Print the number of nodes trimmed between phases. (BCP\_tm\_trim\_tree\_wrapper)
- TmVerb\_FinalStatistics** Print statistics: running time, tree size, best solution value. (For the best solution set TmVerb\_BestFeasibleSolution.
- TmVerb\_ReportDefault** Print out a message when the default version of an overridable method is executed. Default: 1.
- TmVerb\_Last** Just a marker for the last TmVerb.

Definition at line 13 of file BCP\_tm\_param.hpp.

## 4.50.2.2 enum BCP\_tm\_par::int\_params

Integer parameters.

Enumerator

**WarmstartInfo** Specifies how warmstart information should be stored in the TM. Possible values: BCP\_↵ WarmstartNone, BCP\_WarmstartRoot, BCP\_WarmstartParent. The first is obvious, the second means the root node's warmstart info should be sent out along with each node, the last means the parent's warmstart info should be sent out. Default: BCP\_WarmstartParent

**MaxHeapSize** The maximum size of the memory heap the TM can use. If the TM reaches this limit then it converts an LP process into a TS (storage) process. If positive, it's the number of megabytes. 0 indicates the TM should attempt to find this out. -1 indicates that no TS process should be created. Default: -1.

**TmVerb\_SingleLineInfoFrequency** At every this many search tree node provide a single line info on the progress of the search tree. If <= 0 then never. Default: 0.

**TreeSearchStrategy** Which search tree enumeration strategy should be used. Values: 0 (BCP\_BestFirstSearch), 1 (BCP\_BreadthFirstSearch), 2 (BCP\_DepthFirstSearch). Default: 0

**NiceLevel** How resource-hog the processes should be. Interpretation is system dependent, and the value is passed directly to the renice() function. Usually the bigger the number the less demanding the processes will be.

**LpProcessNum** The number of LP processes that should be spawned.

**CgProcessNum** The number of Cut Generator processes that should be spawned.

**CpProcessNum** The number of Cut Pool processes that should be spawned. Values:

**VgProcessNum** The number of Variable Generator processes that should be spawned.

**VpProcessNum** The number of Variable Pool processes that should be spawned.

**TmTimeout** ???

**LPscheduler\_MaxNodeIdNum** Parameters related to scheduling the LP processes.

**LPscheduler\_MaxSbldNum** Max how many SB nodes should the scheduler give to an LP process.

**LPscheduler\_MinSbldNum** Parameters related to scheduling the LP processes.

Definition at line 103 of file BCP\_tm\_param.hpp.

## 4.50.2.3 enum BCP\_tm\_par::dbl\_params

Double parameters.

Enumerator

**UnconditionalDiveProbability** The probability with which the LP process is directed to dive. Values: Default:

**QualityRatioToAllowDiving\_HasUB** The LP process is allowed to dive if the ratio between the quality (for now the presolved objective value) of the child to be kept and the best quality among the candidate nodes is not larger the this parameter. This ratio is applied if an upper bound already exists. Values: Default:

**QualityRatioToAllowDiving\_NoUB** Same as above, but this value is used if an upper bound does not exist yet. Values: Default:

**Granularity** ??? Values: Default:

**MaxRunTime** Maximum allowed running time.

**TerminationGap\_Absolute** ??? Values: Default:

**TerminationGap\_Relative** ??? Values: Default:

**UpperBound** ??? Values: Default:

**LPscheduler\_OverEstimationStatic** Parameters related to scheduling the LP processes.

Definition at line 152 of file BCP\_tm\_param.hpp.

#### 4.50.2.4 enum BCP\_tm\_par::str\_params

String parameters.

Enumerator

**SaveRootCutsTo** The name of the file where those cuts should be saved that were in the root node in the 0-th phase at the end of processing the root node.

**ReadRootCutsFrom** The name of the file where cuts to be added to the root description should be read ot from.

**ExecutableName** The name of the executable that's running (and that should be spawned on the other processors).

**LogFileName** ???

Definition at line 186 of file BCP\_tm\_param.hpp.

#### 4.50.2.5 enum BCP\_tm\_par::str\_array\_params

???

Enumerator

**LpMachines** ???

**CgMachines** ???

**VgMachines** ???

**CpMachines** ???

**VpMachines** ???

Definition at line 203 of file BCP\_tm\_param.hpp.

The documentation for this struct was generated from the following file:

- BCP\_tm\_param.hpp

## 4.51 BCP\_tm\_prob Class Reference

NO OLD DOC.

```
#include <BCP_tm.hpp>
```

Inheritance diagram for BCP\_tm\_prob:

Collaboration diagram for BCP\_tm\_prob:

### Public Member Functions

#### Constructor and destructor

- **BCP\_tm\_prob** ()
- virtual **~BCP\_tm\_prob** ()

#### Methods to pack/unpack objects

- void **pack\_var** (const [BCP\\_var](#) &var)
- [BCP\\_var](#) \* **unpack\_var\_without\_bcpind** ([BCP\\_buffer](#) &buf)
- int **unpack\_var** ()
- void **pack\_cut** (const [BCP\\_cut](#) &cut)
- [BCP\\_cut](#) \* **unpack\_cut\_without\_bcpind** ([BCP\\_buffer](#) &buf)
- int **unpack\_cut** ()

#### Query methods

- char **param** ([BCP\\_tm\\_par::chr\\_params](#) key) const
- int **param** ([BCP\\_tm\\_par::int\\_params](#) key) const
- double **param** ([BCP\\_tm\\_par::dbl\\_params](#) key) const
- const [BCP\\_string](#) & **param** ([BCP\\_tm\\_par::str\\_params](#) key) const
- const [BCP\\_vec](#)< [BCP\\_string](#) > & **param** ([BCP\\_tm\\_par::str\\_array\\_params](#) key) const
- double **granularity** () const
- bool **has\_ub** () const
- double **ub** () const
- bool **ub** (double new\_ub)
- bool **over\_ub** (const double lb) const

#### Public Attributes

- std::map< int, [BCP\\_tm\\_node](#) \* > [active\\_nodes](#)  
*A map from the process ids to the nodes (what they work on)*
- [BCP\\_vec](#)< [BCP\\_tm\\_node](#) \* > [next\\_phase\\_nodes](#)  
*a vector of nodes to be processed in the next phase*

#### User provided members

- [BCP\\_tm\\_user](#) \* **user**
- [BCP\\_user\\_pack](#) \* **packer**  
*A class that holds the methods about how to pack things.*
- [BCP\\_message\\_environment](#) \* **msg\_env**

#### Statistics in the other processes

- [BCP\\_lp\\_statistics](#) \* **lp\_stat**
- [BCP\\_solution](#) \* **feas\_sol**

#### Parameters

- [BCP\\_parameter\\_set](#)< [BCP\\_tm\\_par](#) > **par**
- [BCP\\_slave\\_params](#) **slave\_pars**

#### Flags

- [BCP\\_tm\\_flags](#) **flags**

#### Message passing related fields

- [BCP\\_buffer](#) **msg\_buf**
- std::vector< int > **ts\_procs**
- std::vector< int > **lp\_procs**
- [BCP\\_scheduler](#) **lp\_scheduler**
- double **root\_node\_sent**

*members to measure how long it took to process the root node.*

- double **root\_node\_received\_**

#### The description of the core of the problem

- [BCP\\_problem\\_core](#) \* **core**
- [BCP\\_problem\\_core\\_change](#) \* **core\_as\_change**

#### Vectors indicating the number of leaf nodes assigned to each CP/VP

- [BCP\\_vec](#)< std::pair< int, int > > **leaves\_per\_cp**
- [BCP\\_vec](#)< std::pair< int, int > > **leaves\_per\_vp**

#### Static Public Attributes

- static double [lb\\_multiplier](#)

*The lower bounds of the unexplored search tree nodes.*

### 4.51.1 Detailed Description

NO OLD DOC.

Definition at line 136 of file BCP\_tm.hpp.

### 4.51.2 Member Data Documentation

#### 4.51.2.1 [BCP\\_user\\_pack](#)\* [BCP\\_tm\\_prob::packer](#)

A class that holds the methods about how to pack things.

Definition at line 153 of file BCP\_tm.hpp.

#### 4.51.2.2 double [BCP\\_tm\\_prob::root\\_node\\_sent\\_](#)

members to measure how long it took to process the root node.

Needed for the scheduler (both are in wallclock)

Definition at line 191 of file BCP\_tm.hpp.

#### 4.51.2.3 double [BCP\\_tm\\_prob::lb\\_multiplier](#) `[static]`

The lower bounds of the unexplored search tree nodes.

To avoid rounding errors the lower bounds are stored as floor(lb\*lb\_multiplier)

Definition at line 198 of file BCP\_tm.hpp.

The documentation for this class was generated from the following file:

- [BCP\\_tm.hpp](#)



## 4.52 BCP\_tm\_stat Class Reference

### 4.52.1 Detailed Description

Definition at line 85 of file BCP\_tm.hpp.

The documentation for this class was generated from the following file:

- BCP\_tm.hpp

## 4.53 BCP\_tm\_user Class Reference

The [BCP\\_tm\\_user](#) class is the base class from which the user can derive a problem specific class to be used in the TM process.

```
#include <BCP_tm_user.hpp>
```

Inheritance diagram for BCP\_tm\_user:

Collaboration diagram for BCP\_tm\_user:

### Public Member Functions

- int [process\\_id](#) () const  
*What is the process id of the current process.*
- void [send\\_message](#) (const int target, const [BCP\\_buffer](#) &buf)  
*Send a message to a particular process.*
- void [broadcast\\_message](#) (const BCP\_process\_t proc\_type, const [BCP\\_buffer](#) &buf)  
*Broadcast the message to all processes of the given type.*
- virtual void [process\\_message](#) ([BCP\\_buffer](#) &buf)  
*Process a message that has been sent by another process' user part to this process' user part.*
- virtual void [display\\_feasible\\_solution](#) (const [BCP\\_solution](#) \*sol)  
*Display a feasible solution.*
- virtual void [display\\_node\\_information](#) ([BCP\\_tree](#) &search\_tree, const [BCP\\_tm\\_node](#) &node)  
*Display user information just before a new node is sent to the LP or diving into a node is acknowledged.*
- virtual void [display\\_node\\_information](#) ([BCP\\_tree](#) &search\_tree, const [BCP\\_tm\\_node](#) &node, bool after\_↵  
processing\_node)  
*Display user information.*
- virtual void [display\\_final\\_information](#) (const [BCP\\_lp\\_statistics](#) &lp\_stat)  
*Display information after BCP finished processing the search tree.*

### Methods to set and get the pointer to the BCP\_tm\_prob

object.

*It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.*

- void [setTmProblemPointer](#) ([BCP\\_tm\\_prob](#) \*ptr)  
*Set the pointer.*
- [BCP\\_tm\\_prob](#) \* [getTmProblemPointer](#) () const  
*Get the pointer.*

### Informational methods for the user.

- double `upper_bound` () const  
*Return what is the best known upper bound (might be BCP\_DBL\_MAX)*
- double `lower_bound` () const  
*Return a global lower bound.*

### Methods to get/set BCP parameters on the fly

- char `get_param` (const `BCP_tm_par::chr_params` key) const
- int `get_param` (const `BCP_tm_par::int_params` key) const
- double `get_param` (const `BCP_tm_par::dbl_params` key) const
- const `BCP_string` & `get_param` (const `BCP_tm_par::str_params` key) const
- void `set_param` (const `BCP_tm_par::chr_params` key, const bool val)
- void `set_param` (const `BCP_tm_par::chr_params` key, const char val)
- void `set_param` (const `BCP_tm_par::int_params` key, const int val)
- void `set_param` (const `BCP_tm_par::dbl_params` key, const double val)
- void `set_param` (const `BCP_tm_par::str_params` key, const char \*val)

### Constructor, Destructor

- `BCP_tm_user` ()
- virtual `~BCP_tm_user` ()  
*Being virtual, the destructor invokes the destructor for the real type of the object being deleted.*

### Packing and unpacking methods

- virtual void `pack_module_data` (`BCP_buffer` &buf, `BCP_process_t` ptype)  
*Pack the initial information (info that the user wants to send over) for the process specified by the last argument.*
- virtual `BCP_solution` \* `unpack_feasible_solution` (`BCP_buffer` &buf)  
*Unpack a MIP feasible solution that was packed by the `BCP_lp_user::pack_feasible_solution()` method.*
- virtual bool `replace_solution` (const `BCP_solution` \*old\_sol, const `BCP_solution` \*new\_sol)  
*Decide whether to replace old\_sol with new\_sol.*

### Initial setup (creating core and root)

- virtual void `initialize_core` (`BCP_vec`< `BCP_var_core` \* > &vars, `BCP_vec`< `BCP_cut_core` \* > &cuts, `BCP_lp_relax` \*&matrix)  
*Create the core of the problem by filling out the last three arguments.*
- virtual void `create_root` (`BCP_vec`< `BCP_var` \* > &added\_vars, `BCP_vec`< `BCP_cut` \* > &added\_cuts, `BCP_user_data` \*&user\_data)  
*Create the set of extra variables and cuts that should be added to the formulation in the root node.*

### Initialize new phase

- virtual void `init_new_phase` (int phase, `BCP_column_generation` &colgen, `CoinSearchTreeBase` \*&candidates)  
*Do whatever initialization is necessary before the phase-th phase.*

### If desired, change the tree (the candidate list) in the search

*tree manager using the `setTree()` method.*

*This method is invoked after every insertion into the candidate list and also whenever a new solution is found. In the latter case `new_solution` is true.*

*The default invokes the `newSolution()` and the `reevaluateSearchStrategy()` methods from `CoinSearchTreeManager`.*

- virtual void `change_candidate_heap` (`CoinSearchTreeManager` &candidates, const bool new\_solution)

### 4.53.1 Detailed Description

The [BCP\\_tm\\_user](#) class is the base class from which the user can derive a problem specific class to be used in the TM process.

In that derived class the user can store data to be used in the methods she overrides. Also that is the object the user must return in the `USER_initialize::tm_init()` method.

There are two kind of methods in the class. The non-virtual methods are helper functions for the built-in defaults, but the user can use them as well. The virtual methods execute steps in the BCP algorithm where the user might want to override the default behavior.

The default implementations fall into three major categories.

- Empty; doesn't do anything and immediately returns. (E.g., [pack\\_module\\_data\(\)](#)).
- There is no reasonable default, so throw an exception. This happens if the parameter settings drive the flow of in a way that BCP can't perform the necessary function. This behavior is correct since such methods are invoked only if the parameter settings drive the flow of the algorithm that way, in which case the user better implement those methods. (At the momemnt there is no such method in TM.)
- A default is given. Frequently there are multiple defaults and parameters govern which one is selected (e.g., [compare\\_tree\\_nodes\(\)](#)).

Definition at line 58 of file `BCP_tm_user.hpp`.

### 4.53.2 Constructor & Destructor Documentation

4.53.2.1 `virtual BCP_tm_user::~BCP_tm_user ( ) [inline],[virtual]`

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 113 of file `BCP_tm_user.hpp`.

### 4.53.3 Member Function Documentation

4.53.3.1 `double BCP_tm_user::lower_bound ( ) const`

Return a global lower bound.

This value is the minimum of the true lower bounds in the candidate list and the true lower bounds of the nodes currently processed

4.53.3.2 `virtual void BCP_tm_user::pack_module_data ( BCP_buffer & buf, BCP_process_t ptype ) [virtual]`

Pack the initial information (info that the user wants to send over) for the process specified by the last argument.

The information packed here will be unpacked in the `unpack_module_data()` method of the user defined class in the appropriate process.

Default: empty method.

4.53.3.3 `virtual BCP_solution* BCP_tm_user::unpack_feasible_solution ( BCP_buffer & buf ) [virtual]`

Unpack a MIP feasible solution that was packed by the [BCP\\_lp\\_user::pack\\_feasible\\_solution\(\)](#) method.

Default: Unpacks a [BCP\\_solution\\_generic](#) object. The built-in default should be used if and only if the built-in default was used in [BCP\\_lp\\_user::pack\\_feasible\\_solution\(\)](#).

**4.53.3.4** `virtual bool BCP_tm_user::replace_solution ( const BCP_solution * old_sol, const BCP_solution * new_sol )  
[virtual]`

Decide whether to replace old\_sol with new\_sol.

When this method is invoked it has already been tested that they have the same objective function value. The purpose of the method is that the user can have a secondary objective function.

**4.53.3.5** `virtual void BCP_tm_user::process_message ( BCP_buffer & buf ) [virtual]`

Process a message that has been sent by another process' user part to this process' user part.

**4.53.3.6** `virtual void BCP_tm_user::initialize_core ( BCP_vec< BCP_var_core * > & vars, BCP_vec< BCP_cut_core * >  
& cuts, BCP_lp_relax *& matrix ) [virtual]`

Create the core of the problem by filling out the last three arguments.

These variables/cuts will always stay in the LP relaxation and the corresponding matrix is described by the specified matrix. If there is no core variable or cut then the returned pointer for to the matrix should be a null pointer.

Default: empty method, meaning that there are no variables/cuts in the core and this the core matrix is empty (0 pointer) as well.

**4.53.3.7** `virtual void BCP_tm_user::create_root ( BCP_vec< BCP_var * > & added_vars, BCP_vec< BCP_cut * > &  
added_cuts, BCP_user_data *& user_data ) [virtual]`

Create the set of extra variables and cuts that should be added to the formulation in the root node.

Also decide how variable pricing should be done, that is, if column generation is requested in the [init\\_new\\_phase\(\)](#) method of this class then column generation should be performed according to `pricing_status`.

Default: empty method, meaning that no variables/cuts are added, there is no user data and no pricing should be done.

**4.53.3.8** `virtual void BCP_tm_user::display_node_information ( BCP_tree & search_tree, const BCP_tm_node & node )  
[virtual]`

Display user information just before a new node is sent to the LP or diving into a node is acknowledged.

This method is deprecated in favor of the one with 3 args.

**4.53.3.9** `virtual void BCP_tm_user::display_node_information ( BCP_tree & search_tree, const BCP_tm_node & node, bool  
after_processing_node ) [virtual]`

Display user information.

This method is called just before a node is sent ot for processing (or diving into the node is acknowledged) and just after a node description has been received.

4.53.3.10 `virtual void BCP_tm_user::display_final_information ( const BCP_lp_statistics & lp_stat ) [virtual]`

Display information after BCP finished processing the search tree.

4.53.3.11 `virtual void BCP_tm_user::init_new_phase ( int phase, BCP_column_generation & colgen, CoinSearchTreeBase *& candidates ) [virtual]`

Do whatever initialization is necessary before the `phase`-th phase.

(E.g., setting the pricing strategy.)

The documentation for this class was generated from the following file:

- BCP\_tm\_user.hpp

## 4.54 BCP\_tree Class Reference

NO OLD DOC.

```
#include <BCP_tm_node.hpp>
```

### Public Member Functions

#### Constructor and destructor

- `BCP_tree ()`
- `~BCP_tree ()`

#### Query methods

- `BCP_vec< BCP_tm_node * >::iterator begin ()`
- `BCP_vec< BCP_tm_node * >::iterator end ()`
- `BCP_tm_node * root ()`
- `BCP_tm_node * operator[] (int index)`
- `size_t size () const`
- `int maxdepth () const`
- `int processed () const`
- `void increase_processed ()`

#### Modifying methods

- `double true_lower_bound (const BCP_tm_node *node) const`  
*Return the worst true lower bound in the search tree.*
- `void enumerate_leaves (BCP_tm_node *node, const double obj_limit)`
- `void insert (BCP_tm_node *node)`
- `void remove (int index)`

### 4.54.1 Detailed Description

NO OLD DOC.

Definition at line 178 of file BCP\_tm\_node.hpp.

The documentation for this class was generated from the following file:

- BCP\_tm\_node.hpp

## 4.55 BCP\_ts\_node\_data Struct Reference

Same as [BCP\\_tm\\_node\\_data](#), just there's no need to use smart pointers in this process.

```
#include <BCP_tmstorage.hpp>
```

Collaboration diagram for BCP\_ts\_node\_data:

### 4.55.1 Detailed Description

Same as [BCP\\_tm\\_node\\_data](#), just there's no need to use smart pointers in this process.

Definition at line 67 of file BCP\_tmstorage.hpp.

The documentation for this struct was generated from the following file:

- BCP\_tmstorage.hpp

## 4.56 BCP\_ts\_par Struct Reference

Inheritance diagram for BCP\_ts\_par:

### Public Types

- enum [int\\_params](#) { [MaxHeapSize](#) }

### 4.56.1 Detailed Description

Definition at line 30 of file BCP\_tmstorage.hpp.

### 4.56.2 Member Enumeration Documentation

#### 4.56.2.1 enum BCP\_ts\_par::int\_params

##### Enumerator

**MaxHeapSize** The maximum size of the memory heap the TS can use. The amount of free memory the TS indicates towards the TM uses this value. If positive, it's the number of megabytes. 0 indicates the TS should attempt to find this out. -1 indicates that the TS can use as much memory as it wants. Default: 0.

Definition at line 37 of file BCP\_tmstorage.hpp.

The documentation for this struct was generated from the following file:

- BCP\_tmstorage.hpp

## 4.57 BCP\_ts\_prob Class Reference

Inheritance diagram for BCP\_ts\_prob:

Collaboration diagram for BCP\_ts\_prob:

## Public Attributes

- [BCP\\_vec](#)< int > [indices](#)  
*a list of indices of nodes/vars/cuts that are requested/tobe deleted*
- [BCP\\_vec](#)< int > [positions](#)  
*positions in the TM of requested nodes/vars/cuts*

### 4.57.1 Detailed Description

Definition at line 74 of file BCP\_tmstorage.hpp.

The documentation for this class was generated from the following file:

- BCP\_tmstorage.hpp

## 4.58 BCP\_ts\_user Class Reference

Inheritance diagram for BCP\_ts\_user:

Collaboration diagram for BCP\_ts\_user:

## Public Member Functions

- virtual void [unpack\\_module\\_data](#) ([BCP\\_buffer](#) &buf)  
*Unpack the initial information sent to the LP process by the Tree Manager.*

### Methods to set and get the pointer to the BCP\_ts\_prob

*object.*

*It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.*

- void [setTsProblemPointer](#) ([BCP\\_ts\\_prob](#) \*ptr)  
*Set the pointer.*
- [BCP\\_ts\\_prob](#) \* [getTsProblemPointer](#) ()  
*Get the pointer.*

### Methods to get/set BCP parameters on the fly

- char [get\\_param](#) (const [BCP\\_ts\\_par::chr\\_params](#) key) const
- int [get\\_param](#) (const [BCP\\_ts\\_par::int\\_params](#) key) const
- double [get\\_param](#) (const [BCP\\_ts\\_par::dbl\\_params](#) key) const
- const [BCP\\_string](#) & [get\\_param](#) (const [BCP\\_ts\\_par::str\\_params](#) key) const
- void [set\\_param](#) (const [BCP\\_ts\\_par::chr\\_params](#) key, const bool val)
- void [set\\_param](#) (const [BCP\\_ts\\_par::chr\\_params](#) key, const char val)
- void [set\\_param](#) (const [BCP\\_ts\\_par::int\\_params](#) key, const int val)
- void [set\\_param](#) (const [BCP\\_ts\\_par::dbl\\_params](#) key, const double val)
- void [set\\_param](#) (const [BCP\\_ts\\_par::str\\_params](#) key, const char \*val)

### Constructor, Destructor

- [BCP\\_ts\\_user](#) ()
- virtual [~BCP\\_ts\\_user](#) ()  
*Being virtual, the destructor invokes the destructor for the real type of the object being deleted.*

### 4.58.1 Detailed Description

Definition at line 125 of file BCP\_tmstorage.hpp.

### 4.58.2 Constructor & Destructor Documentation

4.58.2.1 `virtual BCP_ts_user::~~BCP_ts_user ( ) [inline], [virtual]`

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 169 of file BCP\_tmstorage.hpp.

### 4.58.3 Member Function Documentation

4.58.3.1 `virtual void BCP_ts_user::unpack_module_data ( BCP_buffer & buf ) [virtual]`

Unpack the initial information sent to the LP process by the Tree Manager.

This information was packed by the method `BCP_ts_user::pack_module_data()` invoked with `BCP_ProcessType←_TS` as the third (target process type) argument.

Default: empty method.

The documentation for this class was generated from the following file:

- BCP\_tmstorage.hpp

## 4.59 BCP\_user\_class Class Reference

Inheritance diagram for BCP\_user\_class:

### 4.59.1 Detailed Description

Definition at line 26 of file BCP\_USER.hpp.

The documentation for this class was generated from the following file:

- BCP\_USER.hpp

## 4.60 BCP\_user\_data Class Reference

Inheritance diagram for BCP\_user\_data:

Collaboration diagram for BCP\_user\_data:

### 4.60.1 Detailed Description

Definition at line 19 of file BCP\_USER.hpp.

The documentation for this class was generated from the following file:



- BCP\_USER.hpp

## 4.61 BCP\_user\_pack Class Reference

Collaboration diagram for BCP\_user\_pack:

### Public Member Functions

- virtual void [pack\\_warmstart](#) (const [BCP\\_warmstart](#) \*ws, [BCP\\_buffer](#) &buf, bool report\_if\_default=false)  
*Pack warmstarting information.*
- virtual [BCP\\_warmstart](#) \* [unpack\\_warmstart](#) ([BCP\\_buffer](#) &buf, bool report\_if\_default=false)  
*Unpack warmstarting information.*
- virtual void [pack\\_var\\_algo](#) (const [BCP\\_var\\_algo](#) \*var, [BCP\\_buffer](#) &buf)  
*Pack an algorithmic variable.*
- virtual [BCP\\_var\\_algo](#) \* [unpack\\_var\\_algo](#) ([BCP\\_buffer](#) &buf)  
*Unpack an algorithmic variable.*
- virtual void [pack\\_cut\\_algo](#) (const [BCP\\_cut\\_algo](#) \*cut, [BCP\\_buffer](#) &buf)  
*Pack an algorithmic cut.*
- virtual [BCP\\_cut\\_algo](#) \* [unpack\\_cut\\_algo](#) ([BCP\\_buffer](#) &buf)  
*Unpack an algorithmic cut.*
- virtual void [pack\\_user\\_data](#) (const [BCP\\_user\\_data](#) \*ud, [BCP\\_buffer](#) &buf)  
*Pack an user data.*
- virtual [BCP\\_user\\_data](#) \* [unpack\\_user\\_data](#) ([BCP\\_buffer](#) &buf)  
*Unpack an user data.*

### Public Attributes

- [BCP\\_user\\_class](#) \* [user\\_class](#)  
*A pointer of the usr class of the process from which the methods of this class are invoked from.*

#### 4.61.1 Detailed Description

Definition at line 48 of file BCP\_USER.hpp.

#### 4.61.2 Member Data Documentation

##### 4.61.2.1 [BCP\\_user\\_class](#)\* [BCP\\_user\\_pack::user\\_class](#)

A pointer of the usr class of the process from which the methods of this class are invoked from.

The user can try to dynamic cast [user\\_class](#) into [BCP\\_tm\\_user\\*](#), [BCP\\_lp\\_user\\*](#), etc., and when the cast succeeds the user knows in which process the methods was called, and also, the user will have access to the other methods of that class.

Definition at line 55 of file BCP\_USER.hpp.

The documentation for this class was generated from the following file:

- BCP\_USER.hpp

## 4.62 BCP\_var Class Reference

Abstract base class that defines members common to all types of variables.

```
#include <BCP_var.hpp>
```

Inheritance diagram for BCP\_var:

Collaboration diagram for BCP\_var:

### Public Member Functions

#### Constructor and destructor

*Note that there is no default constructor.*

*There is no such thing as "default variable".*

- `BCP_var` (const `BCP_var_t` `var_type`, const double `obj`, const double `lb`, const double `ub`)  
*The constructor sets the internal index of the variable to zero and the other data members to the given arguments.*
- virtual `~BCP_var` ()  
*The destructor is virtual so that the appropriate destructor is invoked for every variable.*

#### Query methods

- virtual `BCP_object_t` `obj_type` () const =0  
*Return the type of the variable.*
- `BCP_var_t` `var_type` () const  
*Return the integrality type of the variable.*
- double `obj` () const  
*Return the objective coefficient.*
- double `lb` () const  
*Return the lower bound.*
- double `ub` () const  
*Return the upper bound.*
- int `bcpind` () const  
*Return the internal index of the variable.*

#### Query methods about the status of the variable

- `BCP_obj_status` `status` () const  
*Return the status of the variable.*
- bool `dont_send_to_pool` () const  
*Return whether the variable should be sent to the Variable Pool process.*
- bool `is_fixed` () const  
*Return whether the variable is fixed or not.*
- bool `is_fixed_to_zero` () const  
*Return whether the variable is fixed to zero or not.*
- bool `is_non_removable` () const  
*Return whether the variable is marked NotRemovable.*
- bool `is_removable` () const  
*Return whether the variable is removable from the formulation at the time of the query.*
- bool `is_to_be_removed` () const  
*Return whether the variable must be removed from the formulation.*

#### Modifying methods

- void `test_inactive` ()  
*Test (and set) whether the var is fixed (inactive)*
- void `set_var_type` (const BCP\_var\_t type)  
*Set the integrality type of the variable.*
- void `set_obj` (const double obj)  
*Set the objective coefficient.*
- void `set_lb` (const double lb)  
*Set the lower bound.*
- void `set_ub` (const double ub)  
*Set the upper bound.*
- void `set_lb_ub` (const double lb, const double ub)  
*Set both lower and upper bounds.*
- void `change_lb_ub_st` (const BCP\_obj\_change &change)  
*Set the lower/upper bounds and the status of the variable simultaneously to the values given in the data members of the argument.*
- void `change_bounds` (const double lb, const double ub)  
*Change the lower and upper bounds to the given values.*
- void `set_bcpind` (const int bcpind)  
*Set the internal index of the variable.*
- void `set_bcpind_flip` ()  
*Flip the internal index of the variable to its negative.*

#### Status modifying methods

- void `set_status` (const BCP\_obj\_status status)  
*Set the status of the variable.*
- void `dont_send_to_pool` (bool flag)  
*Set/unset the flag controlling whether the variable could be sent to the Variable Pool process.*
- void `make_active` ()  
*Mark the variable as active.*
- void `make_non_removable` ()  
*Mark the variable as NotRemovable.*
- void `make_to_be_removed` ()  
*Mark the variable as ToBeRemoved.*

#### Display

- void `display` (const double val) const  
*Display the object type, internal index, and the value given in the argument.*

### Protected Attributes

#### Protected data members

- BCP\_var\_t `_var_type`  
*The integrality type of the variable.*
- double `_obj`  
*The objective coefficient.*
- double `_lb`  
*Lower bound on the value the variable can take.*
- double `_ub`  
*Upper bound on the value the variable can take.*

### 4.62.1 Detailed Description

Abstract base class that defines members common to all types of variables.

Classes describing the three types of variables ([core](#) and [algorithmic](#)) are derived from this class. No object of type [BCP\\_var](#) can exist (having purely virtual members in the class enforces this restriction).

Definition at line 28 of file [BCP\\_var.hpp](#).

### 4.62.2 Constructor & Destructor Documentation

4.62.2.1 `BCP_var::BCP_var ( const BCP_var_t var_type, const double obj, const double lb, const double ub )` `[inline]`

The constructor sets the internal index of the variable to zero and the other data members to the given arguments.

Definition at line 71 of file [BCP\\_var.hpp](#).

4.62.2.2 `virtual BCP_var::~BCP_var ( )` `[inline],[virtual]`

The destructor is virtual so that the appropriate destructor is invoked for every variable.

Definition at line 77 of file [BCP\\_var.hpp](#).

### 4.62.3 Member Function Documentation

4.62.3.1 `virtual BCP_object_t BCP_var::obj_type ( ) const` `[pure virtual]`

Return the type of the variable.

Implemented in [BCP\\_var\\_algo](#), and [BCP\\_var\\_core](#).

4.62.3.2 `BCP_var_t BCP_var::var_type ( ) const` `[inline]`

Return the integrality type of the variable.

Definition at line 85 of file [BCP\\_var.hpp](#).

4.62.3.3 `double BCP_var::obj ( ) const` `[inline]`

Return the objective coefficient.

Definition at line 87 of file [BCP\\_var.hpp](#).

4.62.3.4 `double BCP_var::lb ( ) const` `[inline]`

Return the lower bound.

Definition at line 89 of file [BCP\\_var.hpp](#).

4.62.3.5 `double BCP_var::ub ( ) const` `[inline]`

Return the upper bound.

Definition at line 91 of file BCP\_var.hpp.

**4.62.3.6** `int BCP_var::bcpind ( ) const [inline]`

Return the internal index of the variable.

Definition at line 93 of file BCP\_var.hpp.

**4.62.3.7** `BCP_obj_status BCP_var::status ( ) const [inline]`

Return the status of the variable.

Definition at line 98 of file BCP\_var.hpp.

**4.62.3.8** `bool BCP_var::dont_send_to_pool ( ) const [inline]`

Return whether the variable should be sent to the Variable Pool process.

(Assuming that it stays in the formulation long enough to qualify to be sent to the Variable Pool at all.

Definition at line 102 of file BCP\_var.hpp.

**4.62.3.9** `bool BCP_var::is_fixed ( ) const [inline]`

Return whether the variable is fixed or not.

Definition at line 106 of file BCP\_var.hpp.

**4.62.3.10** `bool BCP_var::is_fixed_to_zero ( ) const [inline]`

Return whether the variable is fixed to zero or not.

Definition at line 110 of file BCP\_var.hpp.

**4.62.3.11** `bool BCP_var::is_non_removable ( ) const [inline]`

Return whether the variable is marked NotRemovable.

Examples of such variables include ???

Definition at line 115 of file BCP\_var.hpp.

**4.62.3.12** `bool BCP_var::is_removable ( ) const [inline]`

Return whether the variable is removable from the formulation at the time of the query.

(It is if it is not non\_removable and it is fixed to zero.)

Definition at line 121 of file BCP\_var.hpp.

**4.62.3.13** `bool BCP_var::is_to_be_removed ( ) const [inline]`

Return whether the variable must be removed from the formulation.

There are very few circumstances when this flag is set; all of them are completely internal to BCP.

Definition at line 127 of file BCP\_var.hpp.

**4.62.3.14** `void BCP_var::set_var_type ( const BCP_var_t type )` `[inline]`

Set the integrality type of the variable.

Definition at line 141 of file BCP\_var.hpp.

**4.62.3.15** `void BCP_var::set_obj ( const double obj )` `[inline]`

Set the objective coefficient.

Definition at line 143 of file BCP\_var.hpp.

**4.62.3.16** `void BCP_var::set_lb ( const double lb )` `[inline]`

Set the lower bound.

Definition at line 145 of file BCP\_var.hpp.

**4.62.3.17** `void BCP_var::set_ub ( const double ub )` `[inline]`

Set the upper bound.

Definition at line 150 of file BCP\_var.hpp.

**4.62.3.18** `void BCP_var::set_lb_ub ( const double lb, const double ub )` `[inline]`

Set both lower and upper bounds.

Definition at line 155 of file BCP\_var.hpp.

**4.62.3.19** `void BCP_var::change_bounds ( const double lb, const double ub )` `[inline]`

Change the lower and upper bounds to the given values.

Definition at line 170 of file BCP\_var.hpp.

**4.62.3.20** `void BCP_var::set_bcpind ( const int bcpind )` `[inline]`

Set the internal index of the variable.

Definition at line 176 of file BCP\_var.hpp.

**4.62.3.21** `void BCP_var::set_status ( const BCP_obj_status status )` `[inline]`

Set the status of the variable.

Definition at line 183 of file BCP\_var.hpp.

**4.62.3.22** void BCP\_var::dont\_send\_to\_pool ( bool *flag* ) [inline]

Set/unset the flag controlling whether the variable could be sent to the Variable Pool process.

Definition at line 186 of file BCP\_var.hpp.

**4.62.3.23** void BCP\_var::make\_active ( ) [inline]

Mark the variable as active.

Note that when this method is invoked the lp formulation must be modified as well: the original bounds of the variable must be reset.

Definition at line 195 of file BCP\_var.hpp.

**4.62.3.24** void BCP\_var::make\_non\_removable ( ) [inline]

Mark the variable as NotRemovable.

Definition at line 199 of file BCP\_var.hpp.

**4.62.3.25** void BCP\_var::make\_to\_be\_removed ( ) [inline]

Mark the variable as ToBeRemoved.

It will actually be removed immediately after all variables that have to be marked this way are marked.

Definition at line 207 of file BCP\_var.hpp.

**4.62.3.26** void BCP\_var::display ( const double *val* ) const

Display the object type, internal index, and the value given in the argument.

(This value is usually the variable's value in an LP solution.)

**4.62.4 Member Data Documentation****4.62.4.1** BCP\_var\_t BCP\_var::\_var\_type [protected]

The integrality type of the variable.

Definition at line 55 of file BCP\_var.hpp.

**4.62.4.2** double BCP\_var::\_obj [protected]

The objective coefficient.

Definition at line 57 of file BCP\_var.hpp.

**4.62.4.3** double BCP\_var::\_lb [protected]

Lower bound on the value the variable can take.

Definition at line 59 of file BCP\_var.hpp.

4.62.4.4 `double BCP_var::_ub` `[protected]`

Upper bound on the value the variable can take.

Definition at line 61 of file `BCP_var.hpp`.

The documentation for this class was generated from the following file:

- `BCP_var.hpp`

## 4.63 BCP\_var\_algo Class Reference

This is the class from which the user should derive her own algorithmic variables.

```
#include <BCP_var.hpp>
```

Inheritance diagram for `BCP_var_algo`:

Collaboration diagram for `BCP_var_algo`:

### Public Member Functions

#### Constructor and destructor

- `BCP_var_algo` (const `BCP_var_t` `var_type`, const double `obj`, const double `lb`, const double `ub`)  
*This constructor just sets the data members to the given values.*
- virtual `~BCP_var_algo` ()=0  
*The destructor deletes the object.*

#### Query methods

- `BCP_object_t` `obj_type` () const  
*Return `BCP_AlgoObj` indicating that the object is an algorithmic variable.*

### Additional Inherited Members

#### 4.63.1 Detailed Description

This is the class from which the user should derive her own algorithmic variables.

Note that such an object cannot be constructed (it has pure virtual methods), only objects with types derived from `BCP_var_algo` can be created. Such objects are constructed either directly by the user or by the unpacking functions of the `BCP_xx_user` classes.

Definition at line 277 of file `BCP_var.hpp`.

#### 4.63.2 Constructor & Destructor Documentation

4.63.2.1 `BCP_var_algo::BCP_var_algo` ( const `BCP_var_t` `var_type`, const double `obj`, const double `lb`, const double `ub` )  
`[inline]`

This constructor just sets the data members to the given values.

See also the constructor of `BCP_var`.

Definition at line 294 of file `BCP_var.hpp`.



4.63.2.2 `virtual BCP_var_algo::~~BCP_var_algo ( ) [pure virtual]`

The destructor deletes the object.

### 4.63.3 Member Function Documentation

4.63.3.1 `BCP_object_t BCP_var_algo::obj_type ( ) const [inline],[virtual]`

Return `BCP_AlgoObj` indicating that the object is an algorithmic variable.

Implements [BCP\\_var](#).

Definition at line 305 of file `BCP_var.hpp`.

The documentation for this class was generated from the following file:

- `BCP_var.hpp`

## 4.64 BCP\_var\_core Class Reference

Core variables are the variables that always stay in the LP formulation.

```
#include <BCP_var.hpp>
```

Inheritance diagram for `BCP_var_core`:

Collaboration diagram for `BCP_var_core`:

### Public Member Functions

#### Constructors and destructor

- [BCP\\_var\\_core](#) (const [BCP\\_var\\_core](#) &x)  
*The copy constructor makes a replica of the argument.*
- [BCP\\_var\\_core](#) (const [BCP\\_var\\_t](#) var\_type, const double obj, const double lb, const double ub)  
*This constructor just sets the data members to the given values.*
- [~BCP\\_var\\_core](#) ()  
*The destructor deletes the object.*

#### Query methods

- `BCP_object_t obj_type () const`  
*Return `BCP_CoreObj` indicating that the object is a core variable.*

### Additional Inherited Members

#### 4.64.1 Detailed Description

Core variables are the variables that always stay in the LP formulation.

Therefore the data members in the base class are quite sufficient to describe the variable. The only thing that has to be done here is overriding the pure virtual method [obj\\_type\(\)](#).

Definition at line 230 of file `BCP_var.hpp`.

### 4.64.2 Constructor & Destructor Documentation

#### 4.64.2.1 `BCP_var_core::BCP_var_core ( const BCP_var_core & x ) [inline]`

The copy constructor makes a replica of the argument.

Definition at line 244 of file `BCP_var.hpp`.

#### 4.64.2.2 `BCP_var_core::BCP_var_core ( const BCP_var_t var_type, const double obj, const double lb, const double ub ) [inline]`

This constructor just sets the data members to the given values.

See also the constructor [BCP\\_var](#).

Definition at line 251 of file `BCP_var.hpp`.

#### 4.64.2.3 `BCP_var_core::~BCP_var_core ( ) [inline]`

The destructor deletes the object.

Definition at line 255 of file `BCP_var.hpp`.

### 4.64.3 Member Function Documentation

#### 4.64.3.1 `BCP_object_t BCP_var_core::obj_type ( ) const [inline],[virtual]`

Return `BCP_CoreObj` indicating that the object is a core variable.

Implements [BCP\\_var](#).

Definition at line 261 of file `BCP_var.hpp`.

The documentation for this class was generated from the following file:

- `BCP_var.hpp`

## 4.65 BCP\_var\_set Class Reference

This class is just a collection of pointers to variables with a number of methods to manipulate these variables and/or select certain entries.

```
#include <BCP_var.hpp>
```

Inheritance diagram for `BCP_var_set`:

Collaboration diagram for `BCP_var_set`:

### Public Member Functions

#### Constructor and destructor

- [BCP\\_var\\_set \( \)](#)

*The default constructor creates a variable set with no variables in it.*

- `~BCP_var_set()`  
*The destructor empties the variable set.*

### Modifying methods

- void `append` (const `BCP_vec< BCP_var * >` &x)  
*Append the variables in the vector `x` to the end of the variable set.*
- void `append` (`BCP_var_set::const_iterator` first, `BCP_var_set::const_iterator` last)  
*Append the variables in `[first, last)` to the end of the variable set.*
- void `set_lb_ub` (const `BCP_vec< int >` &pos, `BCP_vec< double >::const_iterator` bounds)  
*Set the lower/upper bound pairs of the entries given by the contents of `pos` to the values in `[bounds, bounds + pos)`.*
- void `set_lb_ub_st` (const `BCP_vec< BCP_obj_change >` &vc)  
*Set the lower/upper bound pairs and the stati of the first `cc`.*
- void `set_lb_ub_st` (`BCP_vec< int >::const_iterator` pos, const `BCP_vec< BCP_obj_change >` &vc)  
*Set the lower/upper bound pairs and the stati of the entries given by the content of `[pos, pos + cc)`.*

### Methods related to deleting variables from the variable set

- void `deletable` (const int bvarnum, `BCP_vec< int >` &collection)  
*Collect the indices of the variables marked to be removed.*

## Additional Inherited Members

### 4.65.1 Detailed Description

This class is just a collection of pointers to variables with a number of methods to manipulate these variables and/or select certain entries.

Definition at line 316 of file `BCP_var.hpp`.

### 4.65.2 Constructor & Destructor Documentation

#### 4.65.2.1 `BCP_var_set::~~BCP_var_set()` [inline]

The destructor empties the variable set.

*NOTE:* the destructor does NOT delete the variables the members of the variable set point to.

Definition at line 335 of file `BCP_var.hpp`.

### 4.65.3 Member Function Documentation

#### 4.65.3.1 `void BCP_var_set::append ( const BCP_vec< BCP_var * > &x )` [inline]

Append the variables in the vector `x` to the end of the variable set.

Definition at line 342 of file `BCP_var.hpp`.

#### 4.65.3.2 `void BCP_var_set::append ( BCP_var_set::const_iterator first, BCP_var_set::const_iterator last )` [inline]

Append the variables in `[first, last)` to the end of the variable set.

Definition at line 347 of file `BCP_var.hpp`.

4.65.3.3 `void BCP_var_set::set_lb_ub ( const BCP_vec< int > & pos, BCP_vec< double >::const_iterator bounds )`

Set the lower/upper bound pairs of the entries given by the contents of `pos` to the values in `[bounds, bounds + pos.`

`size() )`.

4.65.3.4 `void BCP_var_set::set_lb_ub_st ( const BCP_vec< BCP_obj_change > & vc )`

Set the lower/upper bound pairs and the stati of the first `cc`.

`size() )` entries to the triplets given in the vector. This method is invoked when the variable set is all the variables in the current formulation and we want to change the triplets for the core variables, which are at the beginning of that variable set.

4.65.3.5 `void BCP_var_set::set_lb_ub_st ( BCP_vec< int >::const_iterator pos, const BCP_vec< BCP_obj_change > & vc )`

Set the lower/upper bound pairs and the stati of the entries given by the content of `[pos, pos + cc.`

`size() )` to the triplets contained in `cc`.

4.65.3.6 `void BCP_var_set::deletable ( const int bvarnum, BCP_vec< int > & collection )`

Collect the indices of the variables marked to be removed.

Since core variables are never removed, we pass the number of core variables in the first argument to speed up things a little.

The documentation for this class was generated from the following file:

- BCP\_var.hpp

## 4.66 BCP\_vec< T > Class Template Reference

The class `BCP_vec` serves the same purpose as the vector class in the standard template library.

`#include <BCP_vector.hpp>`

Inheritance diagram for `BCP_vec< T >`:

Collaboration diagram for `BCP_vec< T >`:

### Public Types

#### Type definitions (needed for using the STL)

- `typedef size_t size_type`
- `typedef T value_type`
- `typedef T * iterator`
- `typedef const T * const_iterator`
- `typedef T & reference`
- `typedef const T & const_reference`

## Public Member Functions

### Constructors / Destructor

- [BCP\\_vec](#) ()  
*The default constructor initializes the data members as 0 pointers.*
- [BCP\\_vec](#) (const [BCP\\_vec](#)< T > &x)  
*The copy constructor copies over the content of x.*
- [BCP\\_vec](#) (const size\_t n, const\_reference value=T())  
*Construct a [BCP\\_vec](#) with n elements, all initialized with the second argument (or initialized with the default constructor of T if the second argument is missing).*
- [BCP\\_vec](#) (const\_iterator first, const\_iterator last)  
*Construct a [BCP\\_vec](#) by copying the elements from first to last-1.*
- [BCP\\_vec](#) (const T \*x, const size\_t num)  
*Construct a [BCP\\_vec](#) by copying num objects of type T from the memory starting at x.*
- virtual [~BCP\\_vec](#) ()  
*The destructor deallocates the memory allocated for the [BCP\\_vec](#).*

### Query methods

- iterator [begin](#) ()  
*Return an iterator to the beginning of the object.*
- const\_iterator [begin](#) () const  
*Return a const iterator to the beginning of the object.*
- iterator [end](#) ()  
*Return an iterator to the end of the object.*
- const\_iterator [end](#) () const  
*Return a const iterator to the end of the object.*
- iterator [entry](#) (const int i)  
*Return an iterator to the i-th entry.*
- const\_iterator [entry](#) (const int i) const  
*Return a const iterator to the i-th entry.*
- size\_t [index](#) (const\_iterator pos) const  
*Return the index of the entry pointed to by pos.*
- size\_t [size](#) () const  
*Return the current number of entries.*
- size\_t [capacity](#) () const  
*Return the capacity of the object (space allocated for this many entries).*
- bool [empty](#) () const  
*Test if there are any entries in the object.*
- reference [operator\[\]](#) (const size\_t i)  
*Return a reference to the i-th entry.*
- const\_reference [operator\[\]](#) (const size\_t i) const  
*Return a const reference to the i-th entry.*
- reference [front](#) ()  
*Return a reference to the first entry.*
- const\_reference [front](#) () const  
*Return a const reference to the first entry.*
- reference [back](#) ()  
*Return a reference to the last entry.*
- const\_reference [back](#) () const  
*Return a const reference to the last entry.*

### General modifying methods

- void `reserve` (const size\_t n)  
*Reallocate the object to make space for n entries.*
- void `swap` (BCP\_vec< T > &x)  
*Exchange the contents of the object with that of x.*
- BCP\_vec< T > & `operator=` (const BCP\_vec< T > &x)  
*Copy the contents of x into the object and return a reference the the object itself.*
- void `assign` (const void \*x, const size\_t num)  
*Copy num entries of type T starting at the memory location x into the object.*
- void `insert` (iterator position, const void \*first, const size\_t num)  
*Insert num entries starting from memory location first into the vector from position pos.*
- void `insert` (iterator position, const\_iterator first, const\_iterator last)  
*Insert the entries [first, last) into the vector from position pos.*
- void `insert` (iterator position, const size\_t n, const\_reference x)  
*Insert n copies of x into the vector from position pos.*
- iterator `insert` (iterator position, const\_reference x)  
*Insert x (a single entry) into the vector at position pos.*
- void `append` (const BCP\_vec< T > &x)  
*Append the entries in x to the end of the vector.*
- void `append` (const\_iterator first, const\_iterator last)  
*Append the entries [first, last) to the end of the vector.*
- void `push_back` (const\_reference x)  
*Append x to the end of the vector.*
- void `unchecked_push_back` (const\_reference x)  
*Append x to the end of the vector.*
- void `pop_back` ()  
*Delete the last entry.*
- void `clear` ()  
*Delete every entry.*
- void `update` (const BCP\_vec< int > &positions, const BCP\_vec< T > &values)  
*Update those entries listed in positions to the given values.*
- void `unchecked_update` (const BCP\_vec< int > &positions, const BCP\_vec< T > &values)  
*Same as the previous method but without sanity checks.*

### Methods for selectively keeping entries

- void `keep` (iterator pos)  
*Keep only the entry pointed to by pos.*
- void `keep` (iterator first, iterator last)  
*Keep the entries [first, last).*
- void `keep_by_index` (const BCP\_vec< int > &positions)  
*Keep the entries indexed by indices.*
- void `unchecked_keep_by_index` (const BCP\_vec< int > &positions)  
*Same as the previous method but without the sanity checks.*
- void `keep_by_index` (const int \*firstpos, const int \*lastpos)  
*Keep the entries indexed by the values in [firstpos, lastpos).*
- void `unchecked_keep_by_index` (const int \*firstpos, const int \*lastpos)  
*Same as the previous method but without the sanity checks.*

### Methods for selectively erasing entries

- void `erase` (iterator pos)  
*Erase the entry pointed to by pos.*
- void `erase` (iterator first, iterator last)  
*Erase the entries [first, last).*

- void [erase\\_by\\_index](#) (const [BCP\\_vec](#)< int > &positions)  
*Erase the entries indexed by `indices`.*
- void [unchecked\\_erase\\_by\\_index](#) (const [BCP\\_vec](#)< int > &positions)  
*Same as the previous method but without the sanity check.*
- void [erase\\_by\\_index](#) (const int \*firstpos, const int \*lastpos)  
*Like the other `erase_by_index` method (including sanity checks), just the indices of the entries to be erased are given in `[firstpos, lastpos)`.*
- void [unchecked\\_erase\\_by\\_index](#) (const int \*firstpos, const int \*lastpos)  
*Same as the previous method but without the sanity checks.*

## Protected Member Functions

### Internal methods

- iterator [allocate](#) (size\_t len)  
*allocate raw, uninitialized memory for `len` entries.*
- void [deallocate](#) ()  
*Destroy the entries in the vector and free the memory allocated for the vector.*
- void [insert\\_aux](#) (iterator position, const\_reference x)  
*insert `x` into the given `position` in the vector.*

## Protected Attributes

### Data members

- iterator [start](#)  
*Iterator pointing to the beginning of the memory array where the vector is stored.*
- iterator [finish](#)  
*Iterator pointing to right after the last entry in the vector.*
- iterator [end\\_of\\_storage](#)  
*Iterator pointing to right after the last memory location usable by the vector without reallocation.*

## 4.66.1 Detailed Description

```
template<class T>class BCP_vec< T >
```

The class [BCP\\_vec](#) serves the same purpose as the vector class in the standard template library.

The main difference is that while the vector class is *likely* to be implemented as a memory array, [BCP\\_vec](#) is implemented that way. Also, [BCP\\_vec](#) has a number of extra member methods, most of them exist to speed up operations (e.g., there are *unchecked* versions of the insert member methods, i.e., the method does not check whether there is enough space allocated to fit the new elements).

Definition at line 24 of file [BCP\\_vector.hpp](#).

## 4.66.2 Constructor & Destructor Documentation

### 4.66.2.1 `template<class T> BCP_vec< T >::BCP_vec ( )`

The default constructor initializes the data members as 0 pointers.

Definition at line 65 of file [BCP\\_vector\\_general.hpp](#).

**4.66.2.2** `template<class T> BCP_vec< T >::BCP_vec ( const BCP_vec< T > & x )`

The copy constructor copies over the content of `x`.

Definition at line 68 of file `BCP_vector_general.hpp`.

**4.66.2.3** `template<class T> BCP_vec< T >::BCP_vec ( const size_t n, const_reference value = T () )`

Construct a `BCP_vec` with `n` elements, all initialized with the second argument (or initialized with the default constructor of `T` if the second argument is missing).

Definition at line 75 of file `BCP_vector_general.hpp`.

**4.66.2.4** `template<class T> BCP_vec< T >::BCP_vec ( const_iterator first, const_iterator last )`

Construct a `BCP_vec` by copying the elements from `first` to `last-1`.

Definition at line 86 of file `BCP_vector_general.hpp`.

**4.66.2.5** `template<class T> BCP_vec< T >::BCP_vec ( const T * x, const size_t num )`

Construct a `BCP_vec` by copying `num` objects of type `T` from the memory starting at `x`.

Definition at line 97 of file `BCP_vector_general.hpp`.

**4.66.2.6** `template<class T> virtual BCP_vec< T >::~BCP_vec ( ) [inline],[virtual]`

The destructor deallocates the memory allocated for the `BCP_vec`.

Definition at line 93 of file `BCP_vector.hpp`.

### 4.66.3 Member Function Documentation

**4.66.3.1** `template<class T > BCP_vec< T >::iterator BCP_vec< T >::allocate ( size_t len ) [inline],[protected]`

allocate raw, uninitialized memory for `len` entries.

Definition at line 30 of file `BCP_vector_general.hpp`.

**4.66.3.2** `template<class T > void BCP_vec< T >::deallocate ( ) [inline],[protected]`

Destroy the entries in the vector and free the memory allocated for the vector.

Definition at line 35 of file `BCP_vector_general.hpp`.

**4.66.3.3** `template<class T > void BCP_vec< T >::insert_aux ( iterator position, const_reference x ) [protected]`

insert `x` into the given `position` in the vector.

Reallocate the vector if necessary.

Definition at line 43 of file `BCP_vector_general.hpp`.



**4.66.3.4** `template<class T> iterator BCP_vec< T >::begin ( ) [inline]`

Return an iterator to the beginning of the object.

Definition at line 99 of file BCP\_vector.hpp.

**4.66.3.5** `template<class T> const_iterator BCP_vec< T >::begin ( ) const [inline]`

Return a const iterator to the beginning of the object.

Definition at line 101 of file BCP\_vector.hpp.

**4.66.3.6** `template<class T> iterator BCP_vec< T >::end ( ) [inline]`

Return an iterator to the end of the object.

Definition at line 104 of file BCP\_vector.hpp.

**4.66.3.7** `template<class T> const_iterator BCP_vec< T >::end ( ) const [inline]`

Return a const iterator to the end of the object.

Definition at line 106 of file BCP\_vector.hpp.

**4.66.3.8** `template<class T> iterator BCP_vec< T >::entry ( const int i ) [inline]`

Return an iterator to the *i*-th entry.

Definition at line 109 of file BCP\_vector.hpp.

**4.66.3.9** `template<class T> const_iterator BCP_vec< T >::entry ( const int i ) const [inline]`

Return a const iterator to the *i*-th entry.

Definition at line 111 of file BCP\_vector.hpp.

**4.66.3.10** `template<class T> size_t BCP_vec< T >::index ( const_iterator pos ) const [inline]`

Return the index of the entry pointed to by *pos*.

Definition at line 114 of file BCP\_vector.hpp.

**4.66.3.11** `template<class T> size_t BCP_vec< T >::size ( ) const [inline]`

Return the current number of entries.

Definition at line 116 of file BCP\_vector.hpp.

**4.66.3.12** `template<class T> size_t BCP_vec< T >::capacity ( ) const [inline]`

Return the capacity of the object (space allocated for this many entries).

Definition at line 119 of file BCP\_vector.hpp.

4.66.3.13 `template<class T> bool BCP_vec< T >::empty ( ) const [inline]`

Test if there are any entries in the object.

Definition at line 121 of file BCP\_vector.hpp.

4.66.3.14 `template<class T> reference BCP_vec< T >::operator[] ( const size_t i ) [inline]`

Return a reference to the *i*-th entry.

Definition at line 124 of file BCP\_vector.hpp.

4.66.3.15 `template<class T> const_reference BCP_vec< T >::operator[] ( const size_t i ) const [inline]`

Return a const reference to the *i*-th entry.

Definition at line 126 of file BCP\_vector.hpp.

4.66.3.16 `template<class T> reference BCP_vec< T >::front ( ) [inline]`

Return a reference to the first entry.

Definition at line 129 of file BCP\_vector.hpp.

4.66.3.17 `template<class T> const_reference BCP_vec< T >::front ( ) const [inline]`

Return a const reference to the first entry.

Definition at line 131 of file BCP\_vector.hpp.

4.66.3.18 `template<class T> reference BCP_vec< T >::back ( ) [inline]`

Return a reference to the last entry.

Definition at line 133 of file BCP\_vector.hpp.

4.66.3.19 `template<class T> const_reference BCP_vec< T >::back ( ) const [inline]`

Return a const reference to the last entry.

Definition at line 135 of file BCP\_vector.hpp.

4.66.3.20 `template<class T> void BCP_vec< T >::reserve ( const size_t n )`

Reallocate the object to make space for *n* entries.

Definition at line 112 of file BCP\_vector\_general.hpp.

4.66.3.21 `template<class T> void BCP_vec< T >::swap ( BCP_vec< T > & x ) [inline]`

Exchange the contents of the object with that of *x*.

Definition at line 124 of file BCP\_vector\_general.hpp.

4.66.3.22 `template<class T> BCP_vec< T > & BCP_vec< T >::operator= ( const BCP_vec< T > & x )`

Copy the contents of `x` into the object and return a reference the the object itself.

Definition at line 131 of file `BCP_vector_general.hpp`.

4.66.3.23 `template<class T> void BCP_vec< T >::assign ( const void * x, const size_t num )`

Copy `num` entries of type `T` starting at the memory location `x` into the object.

(`x` is a void pointer since it might be located somewhere in a buffer and therefore might not be aligned for type `T` entries.)

Definition at line 157 of file `BCP_vector_general.hpp`.

4.66.3.24 `template<class T> void BCP_vec< T >::insert ( iterator position, const void * first, const size_t num )`

Insert `num` entries starting from memory location `first` into the vector from position `pos`.

Definition at line 176 of file `BCP_vector_general.hpp`.

4.66.3.25 `template<class T> void BCP_vec< T >::insert ( iterator position, const_iterator first, const_iterator last )`

Insert the entries `[first, last)` into the vector from position `pos`.

Definition at line 225 of file `BCP_vector_general.hpp`.

4.66.3.26 `template<class T> void BCP_vec< T >::insert ( iterator position, const size_t n, const_reference x )`

Insert `n` copies of `x` into the vector from position `pos`.

Definition at line 256 of file `BCP_vector_general.hpp`.

4.66.3.27 `template<class T> BCP_vec< T >::iterator BCP_vec< T >::insert ( iterator position, const_reference x )  
[inline]`

Insert `x` (a single entry) into the vector at position `pos`.

Return an iterator to the newly inserted entry.

Definition at line 287 of file `BCP_vector_general.hpp`.

4.66.3.28 `template<class T> void BCP_vec< T >::append ( const BCP_vec< T > & x ) [inline]`

Append the entries in `x` to the end of the vector.

Definition at line 169 of file `BCP_vector.hpp`.

4.66.3.29 `template<class T> void BCP_vec< T >::push_back ( const_reference x ) [inline]`

Append `x` to the end of the vector.

Check if enough space is allocated (reallocate if necessary).

Definition at line 300 of file `BCP_vector_general.hpp`.

4.66.3.30 `template<class T> void BCP_vec< T >::unchecked_push_back ( const_reference x ) [inline]`

Append `x` to the end of the vector.

Does not check if enough space is allocated.

Definition at line 308 of file `BCP_vector_general.hpp`.

4.66.3.31 `template<class T> void BCP_vec< T >::pop_back ( ) [inline]`

Delete the last entry.

Definition at line 313 of file `BCP_vector_general.hpp`.

4.66.3.32 `template<class T> void BCP_vec< T >::clear ( ) [inline]`

Delete every entry.

Definition at line 318 of file `BCP_vector_general.hpp`.

4.66.3.33 `template<class T> void BCP_vec< T >::update ( const BCP_vec< int > & positions, const BCP_vec< T > & values ) [inline]`

Update those entries listed in `positions` to the given `values`.

The two argument vector must be of equal length. Sanity checks are done on the given positions.

Definition at line 336 of file `BCP_vector_general.hpp`.

4.66.3.34 `template<class T> void BCP_vec< T >::unchecked_update ( const BCP_vec< int > & positions, const BCP_vec< T > & values ) [inline]`

Same as the previous method but without sanity checks.

Definition at line 324 of file `BCP_vector_general.hpp`.

4.66.3.35 `template<class T> void BCP_vec< T >::keep ( iterator pos ) [inline]`

Keep only the entry pointed to by `pos`.

Definition at line 347 of file `BCP_vector_general.hpp`.

4.66.3.36 `template<class T> void BCP_vec< T >::keep ( iterator first, iterator last ) [inline]`

Keep the entries `[first, last)`.

Definition at line 354 of file `BCP_vector_general.hpp`.

4.66.3.37 `template<class T> void BCP_vec< T >::keep_by_index ( const BCP_vec< int > & positions ) [inline]`

Keep the entries indexed by `indices`.

Abort if the indices are not in increasing order, if there are duplicate indices or if any of the indices is outside of the range `[0, size())`.

Definition at line 362 of file BCP\_vector\_general.hpp.

```
4.66.3.38  template<class T> void BCP_vec< T >::unchecked_keep_by_index ( const BCP_vec< int > & positions )
           [inline]
```

Same as the previous method but without the sanity checks.

Definition at line 369 of file BCP\_vector\_general.hpp.

```
4.66.3.39  template<class T> void BCP_vec< T >::keep_by_index ( const int * firstpos, const int * lastpos ) [inline]
```

Keep the entries indexed by the values in [firstpos, lastpos).

Abort if the indices are not in increasing order, if there are duplicate indices or if any of the indices is outside of the range [0, size()).

```
4.66.3.40  template<class T> void BCP_vec< T >::unchecked_keep_by_index ( const int * firstpos, const int * lastpos )
```

Same as the previous method but without the sanity checks.

```
4.66.3.41  template<class T> void BCP_vec< T >::erase ( iterator pos ) [inline]
```

Erase the entry pointed to by pos.

Definition at line 399 of file BCP\_vector\_general.hpp.

```
4.66.3.42  template<class T> void BCP_vec< T >::erase ( iterator first, iterator last ) [inline]
```

Erase the entries [first, last).

Definition at line 406 of file BCP\_vector\_general.hpp.

```
4.66.3.43  template<class T> void BCP_vec< T >::erase_by_index ( const BCP_vec< int > & positions ) [inline]
```

Erase the entries indexed by indices.

Abort if the indices are not in increasing order, if there are duplicate indices or if any of the indices is outside of the range [0, size()).

Definition at line 414 of file BCP\_vector\_general.hpp.

```
4.66.3.44  template<class T> void BCP_vec< T >::unchecked_erase_by_index ( const BCP_vec< int > & positions )
           [inline]
```

Same as the previous method but without the sanity check.

Definition at line 421 of file BCP\_vector\_general.hpp.

```
4.66.3.45  template<class T> void BCP_vec< T >::erase_by_index ( const int * firstpos, const int * lastpos ) [inline]
```

Like the other erase\_by\_index method (including sanity checks), just the indices of the entries to be erased are given in [firstpos, lastpos).

4.66.3.46 `template<class T> void BCP_vec< T >::unchecked_erase_by_index ( const int * firstpos, const int * lastpos )`

Same as the previous method but without the sanity checks.

#### 4.66.4 Member Data Documentation

4.66.4.1 `template<class T> iterator BCP_vec< T >::start` `[protected]`

Iterator pointing to the beginning of the memory array where the vector is stored.

Definition at line 66 of file BCP\_vector.hpp.

4.66.4.2 `template<class T> iterator BCP_vec< T >::finish` `[protected]`

Iterator pointing to right after the last entry in the vector.

Definition at line 68 of file BCP\_vector.hpp.

4.66.4.3 `template<class T> iterator BCP_vec< T >::end_of_storage` `[protected]`

Iterator pointing to right after the last memory location usable by the vector without reallocation.

Definition at line 71 of file BCP\_vector.hpp.

The documentation for this class was generated from the following files:

- BCP\_vector.hpp
- BCP\_vector\_general.hpp

### 4.67 BCP\_vec\_change< T > Class Template Reference

This class stores a vector explicitly or relatively to another vector.

`#include <BCP_vector_change.hpp>`

Inheritance diagram for BCP\_vec\_change< T >:

#### Public Member Functions

##### Constructors and destructor

- [BCP\\_vec\\_change](#) (const BCP\_storage\_t st)  
*Construct a vector change that's empty and is of the given storage.*
- [BCP\\_vec\\_change](#) (const T \*first, const T \*last)  
*Construct an explicit description describing the vector bounded by the two iterators.*
- [BCP\\_vec\\_change](#) (const [BCP\\_vec\\_change](#)< T > &old\_vec, const [BCP\\_vec\\_change](#)< T > &new\_vec, const [BCP\\_vec](#)< int > &del\_pos)  
*Construct a relative description.*
- [BCP\\_vec\\_change](#) (const [BCP\\_vec\\_change](#)< T > &old\_vec, const [BCP\\_vec\\_change](#)< T > &new\_vec, const [BCP\\_vec](#)< int > &del\_pos, const double etol)  
*Construct a relative description.*
- [BCP\\_vec\\_change](#) ([BCP\\_buffer](#) &buf)

*Construct the object by unpacking it from a buffer.*

- `~BCP_vec_change ()`

*The destructor need not do anything.*

### Query methods

- `BCP_storage_t storage () const`

*Return the storage type of the vector.*

- `const BCP_vec< T > & explicit_vector () const`

*Return a const reference to the vector if it is explicitly stored, otherwise throw an exception.*

- `int storage_size () const`

*Return how much memory it'll take to pack this info.*

### Modifying methods

- `void update (const BCP_vec_change< T > &change)`

*Update the current vector with the argument vector.*

### Packing and unpacking

- `void pack (BCP_buffer &buf) const`

*Pack the data into a buffer.*

- `void unpack (BCP_buffer &buf)`

*Unpack the data from a buffer.*

## 4.67.1 Detailed Description

```
template<class T>class BCP_vec_change< T >
```

This class stores a vector explicitly or relatively to another vector.

The class can be used when we want to store a sequence of vectors in as small a space as possible. The first vector must be stored explicitly, the others can be either explicit ones or stored with respect to the previous vector.

Definition at line 20 of file BCP\_vector\_change.hpp.

## 4.67.2 Constructor & Destructor Documentation

4.67.2.1 `template<class T> BCP_vec_change< T >::BCP_vec_change ( const BCP_storage_t st ) [inline]`

Construct a vector change that's empty and is of the given storage.

If the storage is explicit then this creates an explicit vector with no entry. If it is relative, then the data describes that there is no change with respect to the parent vector.

Definition at line 43 of file BCP\_vector\_change.hpp.

4.67.2.2 `template<class T> BCP_vec_change< T >::BCP_vec_change ( const T * first, const T * last ) [inline]`

Construct an explicit description describing the vector bounded by the two iterators.

Definition at line 47 of file BCP\_vector\_change.hpp.

4.67.2.3 `template<class T> BCP_vec_change< T >::BCP_vec_change ( const BCP_vec_change< T > & old_vec,  
const BCP_vec_change< T > & new_vec, const BCP_vec< int > & del_pos ) [inline]`

Construct a relative description.



## Parameters

<i>new_vec</i>	the vector that should result after the change is applied to
<i>old_vec</i>	the original vector
<i>del_pos</i>	specifies which entries are to be deleted from <i>old_vec</i> before the change in this object can be applied.

Definition at line 59 of file BCP\_vector\_change.hpp.

**4.67.2.4** `template<class T> BCP_vec_change< T >::BCP_vec_change ( const BCP_vec_change< T > & old_vec, const BCP_vec_change< T > & new_vec, const BCP_vec< int > & del_pos, const double etol ) [inline]`

Construct a relative description.

Same as the previous constructor, except that there is an extra argument, the epsilon tolerance for equality testing. This makes sense only if `is_double`, so this constructor should be used only in that case.

Definition at line 100 of file BCP\_vector\_change.hpp.

**4.67.2.5** `template<class T> BCP_vec_change< T >::BCP_vec_change ( BCP_buffer & buf ) [inline]`

Construct the object by unpacking it from a buffer.

Definition at line 137 of file BCP\_vector\_change.hpp.

**4.67.2.6** `template<class T> BCP_vec_change< T >::~BCP_vec_change ( ) [inline]`

The destructor need not do anything.

Definition at line 142 of file BCP\_vector\_change.hpp.

### 4.67.3 Member Function Documentation

**4.67.3.1** `template<class T> int BCP_vec_change< T >::storage_size ( ) const [inline]`

Return how much memory it'll take to pack this info.

It is used when comparing which sort of storage is smaller

Definition at line 161 of file BCP\_vector\_change.hpp.

**4.67.3.2** `template<class T> void BCP_vec_change< T >::update ( const BCP_vec_change< T > & change ) [inline]`

Update the current vector with the argument vector.

If the argument vector is explicit then just replace the current vector. If it is relative and the current vector is explicit, then perform the update. Otherwise throw an exception (the operation is impossible.)

Definition at line 175 of file BCP\_vector\_change.hpp.

**4.67.3.3** `template<class T> void BCP_vec_change< T >::pack ( BCP_buffer & buf ) const [inline]`

Pack the data into a buffer.

Definition at line 208 of file BCP\_vector\_change.hpp.

4.67.3.4 `template<class T> void BCP_vec_change< T >::unpack ( BCP_buffer & buf ) [inline]`

Unpack the data from a buffer.

Definition at line 213 of file BCP\_vector\_change.hpp.

The documentation for this class was generated from the following file:

- BCP\_vector\_change.hpp

## 4.68 BCP\_vg\_par Struct Reference

Inheritance diagram for BCP\_vg\_par:

### Public Types

- enum `chr_params` { `MessagePassingIsSerial`, `ReportWhenDefaultIsExecuted` }
- enum `int_params` { `NiceLevel` }

### 4.68.1 Detailed Description

Definition at line 6 of file BCP\_vg\_param.hpp.

### 4.68.2 Member Enumeration Documentation

#### 4.68.2.1 enum BCP\_vg\_par::chr\_params

Enumerator

***MessagePassingIsSerial*** Indicates whether message passing is serial (all processes are on the same processor) or not.

Values: true (1), false (0). Default: 1.

***ReportWhenDefaultIsExecuted*** Print out a message when the default version of an overridable method is executed. Default: 1.

Definition at line 7 of file BCP\_vg\_param.hpp.

#### 4.68.2.2 enum BCP\_vg\_par::int\_params

Enumerator

***NiceLevel*** The "nice" level the process should run at. On Linux and on AIX this value should be between -20 and 20. The higher this value the less resource the process will receive from the system. Note that

1. The process starts with 0 priority and it can only be increased.
2. If the load is low on the machine (e.g., when all other processes are interactive like netscape or text editors) then even with 20 priority the process will use close to 100% of the cpu, and the interactive processes will be noticeably more responsive than they would be if this process ran with 0 priority.

See also

the man page of the `setpriority` system function.

Definition at line 18 of file `BCP_vg_param.hpp`.

The documentation for this struct was generated from the following file:

- `BCP_vg_param.hpp`

## 4.69 BCP\_vg\_prob Class Reference

This class is the central class of the Variable Generator process.

```
#include <BCP_vg.hpp>
```

Inheritance diagram for `BCP_vg_prob`:

Collaboration diagram for `BCP_vg_prob`:

### Public Member Functions

- `BCP_cut * unpack_cut ()`

*Unpack a cut.*

### Constructor and destructor

- `BCP_vg_prob (int my_id, int parent)`

*The default constructor.*

- `virtual ~BCP_vg_prob ()`

*The destructor deletes everything.*

### Query methods

- `bool has_ub () const`

*Return true/false indicating whether any upper bound has been found.*

- `double ub () const`

*Return the current upper bound (`BCP_DBL_MAX/10` if there's no upper bound found yet.)*

### Modifying methods

- `void ub (const double bd)`

*Set the upper bound equal to the argument.*

- `bool probe_messages ()`

*Test if there is a message in the message queue waiting to be processed.*

### Public Attributes

#### Data members

- `BCP_vg_user * user`

*The user object holding the user's data.*

- `BCP_user_pack * packer`

- A class that holds the methods about how to pack things.*
- [BCP\\_message\\_environment](#) \* [msg\\_env](#)  
*The message passing environment.*
- [BCP\\_buffer](#) [msg\\_buf](#)  
*The message buffer of the Variable Generator process.*
- [BCP\\_parameter\\_set](#) < [BCP\\_vg\\_par](#) > [par](#)  
*The parameters controlling the Variable Generator process.*
- [BCP\\_problem\\_core](#) \* [core](#)  
*The description of the core of the problem.*
- double [upper\\_bound](#)  
*The proc id of the Tree Manager.*
- [BCP\\_vec](#) < [BCP\\_cut](#) \* > [cuts](#)  
*Variables are to be generated for the LP solution given by these cuts and their values (next member).*
- [BCP\\_vec](#) < double > [pi](#)  
*The dual values corresponding to the cuts above.*
- int [sender](#)  
*The process id of the LP process that sent the solution.*
- int [phase](#)  
*The phase the algorithm is in.*
- int [node\\_level](#)  
*The level of search tree node where the solution was generated.*
- int [node\\_index](#)  
*The index of search tree node where the solution was generated.*
- int [node\\_iteration](#)  
*The iteration within the search tree node where the solution was generated.*

#### 4.69.1 Detailed Description

This class is the central class of the Variable Generator process.

Only one object of this type is created and that holds all the data in the VG process. A reference to that object is passed to (almost) every function (or member method) that's invoked within the VG process.

Definition at line 32 of file `BCP_vg.hpp`.

#### 4.69.2 Constructor & Destructor Documentation

##### 4.69.2.1 `BCP_vg_prob::BCP_vg_prob ( int my_id, int parent )`

The default constructor.

Initializes every data member to a natural state.

##### 4.69.2.2 `virtual BCP_vg_prob::~BCP_vg_prob ( ) [virtual]`

The destructor deletes everything.

#### 4.69.3 Member Function Documentation

##### 4.69.3.1 `bool BCP_vg_prob::has_ub ( ) const [inline]`

Return true/false indicating whether any upper bound has been found.

Definition at line 113 of file `BCP_vg.hpp`.

**4.69.3.2** void BCP\_vg\_prob::ub ( const double *bd* ) [inline]

Set the upper bound equal to the argument.

Definition at line 122 of file BCP\_vg.hpp.

**4.69.3.3** bool BCP\_vg\_prob::probe\_messages ( )

Test if there is a message in the message queue waiting to be processed.

**4.69.3.4** BCP\_cut\* BCP\_vg\_prob::unpack\_cut ( )

Unpack a cut.

Invoked from the built-in [BCP\\_vg\\_user::unpack\\_dual\\_solution\(\)](#).

## 4.69.4 Member Data Documentation

**4.69.4.1** BCP\_vg\_user\* BCP\_vg\_prob::user

The user object holding the user's data.

This object is created by a call to the appropriate member of [\[USER\\_initialize\]](#){[USER\\_initialize.html](#)}.

Definition at line 49 of file BCP\_vg.hpp.

**4.69.4.2** BCP\_user\_pack\* BCP\_vg\_prob::packer

A class that holds the methods about how to pack things.

Definition at line 52 of file BCP\_vg.hpp.

**4.69.4.3** BCP\_message\_environment\* BCP\_vg\_prob::msg\_env

The message passing environment.

This object is created by a call to the appropriate member of [\[USER\\_initialize\]](#){[USER\\_initialize.html](#)}.

Definition at line 57 of file BCP\_vg.hpp.

**4.69.4.4** BCP\_buffer BCP\_vg\_prob::msg\_buf

The message buffer of the Variable Generator process.

Definition at line 60 of file BCP\_vg.hpp.

**4.69.4.5** BCP\_parameter\_set<BCP\_vg\_par> BCP\_vg\_prob::par

The parameters controlling the Variable Generator process.

Definition at line 63 of file BCP\_vg.hpp.

#### 4.69.4.6 **BCP\_problem\_core\*** BCP\_vg\_prob::core

The description of the core of the problem.

Definition at line 66 of file BCP\_vg.hpp.

#### 4.69.4.7 **double** BCP\_vg\_prob::upper\_bound

The proc id of the Tree Manager.

The best currently known upper bound.

Definition at line 72 of file BCP\_vg.hpp.

#### 4.69.4.8 **BCP\_vec<BCP\_cut\*>** BCP\_vg\_prob::cuts

Variables are to be generated for the LP solution given by these cuts and their values (next member).

Not all cuts need to be listed (e.g., list only those that have nonzero dual values in the current solution).

See also

[BCP\\_lp\\_user::pack\\_dual\\_solution\(\)](#)

Definition at line 82 of file BCP\_vg.hpp.

#### 4.69.4.9 **BCP\_vec<double>** BCP\_vg\_prob::pi

The dual values corresponding to the cuts above.

Definition at line 84 of file BCP\_vg.hpp.

#### 4.69.4.10 **int** BCP\_vg\_prob::sender

The process id of the LP process that sent the solution.

Definition at line 86 of file BCP\_vg.hpp.

#### 4.69.4.11 **int** BCP\_vg\_prob::phase

The phase the algorithm is in.

Definition at line 89 of file BCP\_vg.hpp.

#### 4.69.4.12 **int** BCP\_vg\_prob::node\_level

The level of search tree node where the solution was generated.

Definition at line 91 of file BCP\_vg.hpp.

#### 4.69.4.13 **int** BCP\_vg\_prob::node\_index

The index of search tree node where the solution was generated.

Definition at line 93 of file BCP\_vg.hpp.

## 4.69.4.14 int BCP\_vg\_prob::node\_iteration

The iteration within the search tree node where the solution was generated.

Definition at line 96 of file BCP\_vg.hpp.

The documentation for this class was generated from the following file:

- BCP\_vg.hpp

## 4.70 BCP\_vg\_user Class Reference

The [BCP\\_vg\\_user](#) class is the base class from which the user can derive a problem specific class to be used in the Cut Generator process.

```
#include <BCP_vg_user.hpp>
```

Inheritance diagram for BCP\_vg\_user:

Collaboration diagram for BCP\_vg\_user:

### Public Member Functions

- void [send\\_var](#) (const [BCP\\_var](#) &var)  
*Pack the argument into the message buffer and send it to the sender of the LP solution.*
- virtual void [unpack\\_module\\_data](#) ([BCP\\_buffer](#) &buf)  
*Unpack the initial information sent to the Variable Generator process by the Tree Manager.*
- virtual void [unpack\\_dual\\_solution](#) ([BCP\\_buffer](#) &buf)  
*Unpack the LP solution arriving from the LP process.*
- virtual void [generate\\_vars](#) ([BCP\\_vec](#)< [BCP\\_cut](#) \* > &cuts, [BCP\\_vec](#)< double > &pi)  
*Perform the actual variable generation.*

### Methods to set and get the pointer to the BCP\_vg\_prob

object.

It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.

- void [setVgProblemPointer](#) ([BCP\\_vg\\_prob](#) \*ptr)  
*Set the pointer.*
- [BCP\\_vg\\_prob](#) \* [getVgProblemPointer](#) ()  
*Get the pointer.*

### Informational methods for the user.

- double [upper\\_bound](#) () const  
*Return what is the best known upper bound (might be BCP\_DBL\_MAX)*
- int [current\\_phase](#) () const  
*Return the phase the algorithm is in.*
- int [current\\_level](#) () const  
*Return the level of the search tree node for which cuts are being generated.*
- int [current\\_index](#) () const  
*Return the internal index of the search tree node for which cuts are being generated.*

- int [current\\_iteration](#) () const

*Return the iteration count within the search tree node for which cuts are being generated.*

### Methods to get/set BCP parameters on the fly

- char **get\_param** (const [BCP\\_vg\\_par::chr\\_params](#) key) const
- int **get\_param** (const [BCP\\_vg\\_par::int\\_params](#) key) const
- double **get\_param** (const [BCP\\_vg\\_par::dbl\\_params](#) key) const
- const [BCP\\_string](#) & **get\_param** (const [BCP\\_vg\\_par::str\\_params](#) key) const
- void **set\_param** (const [BCP\\_vg\\_par::chr\\_params](#) key, const bool val)
- void **set\_param** (const [BCP\\_vg\\_par::chr\\_params](#) key, const char val)
- void **set\_param** (const [BCP\\_vg\\_par::int\\_params](#) key, const int val)
- void **set\_param** (const [BCP\\_vg\\_par::dbl\\_params](#) key, const double val)
- void **set\_param** (const [BCP\\_vg\\_par::str\\_params](#) key, const char \*val)

### Constructor, Destructor

- **BCP\_vg\_user** ()
- virtual [~BCP\\_vg\\_user](#) ()

*Being virtual, the destructor invokes the destructor for the real type of the object being deleted.*

## 4.70.1 Detailed Description

The [BCP\\_vg\\_user](#) class is the base class from which the user can derive a problem specific class to be used in the Cut Generator process.

In that derived class the user can store data to be used in the methods she overrides. Also that is the object the user must return in the `USER_initialize::vg_init()` method.

There are two kind of methods in the class. The non-virtual methods are helper functions for the built-in defaults, but the user can use them as well. The virtual methods execute steps in the BCP algorithm where the user might want to override the default behavior.

The default implementations fall into three major categories.

- Empty; doesn't do anything and immediately returns (e.g., [unpack\\_module\\_data\(\)](#)). `unpack_module_data.`
- There is no reasonable default, so throw an exception. This happens if the parameter settings drive the flow of in a way that BCP can't perform the necessary function. This behavior is correct since such methods are invoked only if the parameter settings drive the flow of the algorithm that way, in which case the user better implement those methods. (At the momemnt there is no such method in VG.)
- A default is given. Frequently there are multiple defaults and parameters govern which one is selected (e.g., [unpack\\_dual\\_solution\(\)](#)).

Definition at line 51 of file `BCP_vg_user.hpp`.

## 4.70.2 Constructor & Destructor Documentation

### 4.70.2.1 virtual `BCP_vg_user::~~BCP_vg_user ( )` `[inline]`, `[virtual]`

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 120 of file `BCP_vg_user.hpp`.



### 4.70.3 Member Function Documentation

#### 4.70.3.1 void BCP\_vg\_user::send\_var ( const BCP\_var & var )

Pack the argument into the message buffer and send it to the sender of the LP solution.

Whenever the user generates a variable in the [generate\\_vars\(\)](#) method she should invoke this method to immediately send off the variable to the LP process.

#### 4.70.3.2 virtual void BCP\_vg\_user::unpack\_module\_data ( BCP\_buffer & buf ) [virtual]

Unpack the initial information sent to the Variable Generator process by the Tree Manager.

This information was packed by the method [BCP\\_tm\\_user::pack\\_module\\_data\(\)](#) invoked with `BCP_ProcessType←_VG` as the third (target process type) argument.

Default: empty method.

#### 4.70.3.3 virtual void BCP\_vg\_user::unpack\_dual\_solution ( BCP\_buffer & buf ) [virtual]

Unpack the LP solution arriving from the LP process.

This method is invoked only if the user packs the info necessary for variable generation by herself, i.e., she overrides the [BCP\\_lp\\_user::pack\\_dual\\_solution\(\)](#) method. If that's the case the user has to unpack the same info she has packed in the LP process.

#### 4.70.3.4 virtual void BCP\_vg\_user::generate\_vars ( BCP\_vec< BCP\_cut \* > & cuts, BCP\_vec< double > & pi ) [virtual]

Perform the actual variable generation.

Whenever a variable is generated, the user should invoke the [send\\_var\(\)](#) method to send the generated variable back to the LP process.

The documentation for this class was generated from the following file:

- BCP\_vg\_user.hpp

## 4.71 BCP\_warmstart Class Reference

Warmstarting information for the LP solver.

```
#include <BCP_warmstart.hpp>
```

Inheritance diagram for BCP\_warmstart:

### Public Member Functions

- virtual [~BCP\\_warmstart](#) ()  
*The destructor is pure virtual.*
- virtual **CoinWarmStart** \* [convert\\_to\\_CoinWarmStart](#) () const =0  
*Return an OsiWarmStart object that can be fed to the LP engine.*
- virtual BCP\_storage\_t [storage](#) () const =0

*Return how the warmstarting info is stored.*

- virtual void [update](#) (const [BCP\\_warmstart](#) \*const change)=0

*Update the current data with the one in the argument.*

- virtual [BCP\\_warmstart](#) \* [as\\_change](#) (const [BCP\\_warmstart](#) \*const old\_ws, const [BCP\\_vec](#)< int > &del\_vars, const [BCP\\_vec](#)< int > &del\_cuts, const double petol, const double detol) const =0

*Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.*

- virtual [BCP\\_warmstart](#) \* [clone](#) () const =0

*Make a replica of the current warmstart information.*

- virtual [BCP\\_warmstart](#) \* [empty\\_wrt\\_this](#) () const =0

*Create a warmstart info describing that no change should be done.*

- virtual int [storage\\_size](#) () const =0

*Return how much memory it'll take to pack this warmstart info.*

#### 4.71.1 Detailed Description

Warmstarting information for the LP solver.

A realization of the warmstarting information must done in a way that allows to keep the information either in an explicit way or as a change relative to another warmstarting information.

Definition at line 24 of file [BCP\\_warmstart.hpp](#).

#### 4.71.2 Constructor & Destructor Documentation

4.71.2.1 `virtual BCP_warmstart::~BCP_warmstart ( ) [inline],[virtual]`

The destructor is pure virtual.

(There are no data members here.)

Definition at line 27 of file [BCP\\_warmstart.hpp](#).

#### 4.71.3 Member Function Documentation

4.71.3.1 `virtual BCP_storage_t BCP_warmstart::storage ( ) const [pure virtual]`

Return how the warmstarting info is stored.

Implemented in [BCP\\_warmstart\\_basis](#), [BCP\\_warmstart\\_primaldual](#), and [BCP\\_warmstart\\_dual](#).

4.71.3.2 `virtual void BCP_warmstart::update ( const BCP_warmstart *const change ) [pure virtual]`

Update the current data with the one in the argument.

If the argument is of explicit storage then just replace the current data. If it is relative and the current data is explicit then perform the update. Otherwise throw an exception.

Implemented in [BCP\\_warmstart\\_primaldual](#), [BCP\\_warmstart\\_basis](#), and [BCP\\_warmstart\\_dual](#).

4.71.3.3 `virtual BCP_warmstart* BCP_warmstart::as_change ( const BCP_warmstart *const old_ws, const BCP_vec< int > & del_vars, const BCP_vec< int > & del_cuts, const double petol, const double detol ) const [pure virtual]`

Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.

However, if the currently stored data is shorter to store than the change, then this method can return a copy of the current data! The current data must be explicitly stored and `old_ws` must be either explicit or can contain no data. Otherwise an exception must be thrown.

#### Parameters

<i>old_ws</i>	the old warmstart info
<i>del_vars</i>	the indices of the variables that are deleted from the formulation <code>old_ws</code> was created for
<i>del_cuts</i>	same for the cuts
<i>petol</i>	primal zero tolerance
<i>detol</i>	dual zero tolerance

Implemented in [BCP\\_warmstart\\_primaldual](#), [BCP\\_warmstart\\_basis](#), and [BCP\\_warmstart\\_dual](#).

4.71.3.4 `virtual BCP_warmstart* BCP_warmstart::clone ( ) const [pure virtual]`

Make a replica of the current warmstart information.

Implemented in [BCP\\_warmstart\\_primaldual](#), [BCP\\_warmstart\\_basis](#), and [BCP\\_warmstart\\_dual](#).

4.71.3.5 `virtual BCP_warmstart* BCP_warmstart::empty_wrt_this ( ) const [pure virtual]`

Create a warmstart info describing that no change should be done.

This is really the task of a constructor, but BCP does not know the type of warmstart the user will use, so it will invoke this method for a warmstart that was created by the user. Tricky, isn't it?

Implemented in [BCP\\_warmstart\\_primaldual](#), [BCP\\_warmstart\\_basis](#), and [BCP\\_warmstart\\_dual](#).

4.71.3.6 `virtual int BCP_warmstart::storage_size ( ) const [pure virtual]`

Return how much memory it'll take to pack this warmstart info.

It is used when comparing which sort of storage is smaller.

Implemented in [BCP\\_warmstart\\_primaldual](#), [BCP\\_warmstart\\_basis](#), and [BCP\\_warmstart\\_dual](#).

The documentation for this class was generated from the following file:

- `BCP_warmstart.hpp`

## 4.72 BCP\_warmstart\_basis Class Reference

This class describes a warmstart information that consists of basis information for structural and artificial variables.

```
#include <BCP_warmstart_basis.hpp>
```

Inheritance diagram for `BCP_warmstart_basis`:

Collaboration diagram for `BCP_warmstart_basis`:

## Public Member Functions

- virtual **CoinWarmStart** \* [convert\\_to\\_CoinWarmStart](#) () const  
*Return an CoinwarmStart object that can be fed to the LP engine.*
- virtual BCP\_storage\_t [storage](#) () const  
*Return how the warmstarting info is stored.*
- virtual BCP\_warmstart \* [clone](#) () const  
*Make a replica of the current warmstart information.*
- virtual BCP\_warmstart \* [empty\\_wrt\\_this](#) () const  
*Create a warmstart info describing that no change should be done.*
- virtual int [storage\\_size](#) () const  
*Return how much memory it'll take to pack this warmstart info.*
- virtual void [update](#) (const BCP\_warmstart \*const change)  
*Update the current data with the one in the argument.*
- virtual BCP\_warmstart \* [as\\_change](#) (const BCP\_warmstart \*const old\_ws, const BCP\_vec< int > &del\_vars, const BCP\_vec< int > &del\_cuts, const double petol, const double detol) const  
*Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in old\_ws.*
- void [pack](#) (BCP\_buffer &buf) const  
*Pack the warmstart info into a buffer.*

## Constructors and destructor

- [BCP\\_warmstart\\_basis](#) (BCP\_buffer &buf)  
*Create the object by unpacking it from a buffer.*
- [BCP\\_warmstart\\_basis](#) (const char \*vfirst, const char \*vlast, const char \*cfirst, const char \*clast)  
*Create an explicitly stored warmstart info by considering the two character arrays ([vfirst,vlast) and [cfirst,clast)) as the status arrays for the variables/cuts.*
- [BCP\\_warmstart\\_basis](#) (const BCP\_warmstart\_basis &ws)  
*Copy constructor.*
- ~[BCP\\_warmstart\\_basis](#) ()

### 4.72.1 Detailed Description

This class describes a warmstart information that consists of basis information for structural and artificial variables.

"basic / on\_upper / on\_lower" info is stored on 2 bits for every variable. Only the added methods are documented in details on this page, for the inherited methods see the description of the base class.

Definition at line 24 of file BCP\_warmstart\_basis.hpp.

### 4.72.2 Member Function Documentation

#### 4.72.2.1 virtual CoinWarmStart\* BCP\_warmstart\_basis::convert\_to\_CoinWarmStart( ) const [virtual]

Return an CoinwarmStart object that can be fed to the LP engine.

The implementation for this class will return an CoinwarmStartBasis object.

Implements [BCP\\_warmstart](#).

4.72.2.2 `virtual BCP_storage_t BCP_warmstart_basis::storage ( ) const [virtual]`

Return how the warmstarting info is stored.

Implements [BCP\\_warmstart](#).

4.72.2.3 `virtual BCP_warmstart* BCP_warmstart_basis::clone ( ) const [inline],[virtual]`

Make a replica of the current warmstart information.

Implements [BCP\\_warmstart](#).

Definition at line 69 of file `BCP_warmstart_basis.hpp`.

4.72.2.4 `virtual BCP_warmstart* BCP_warmstart_basis::empty_wrt_this ( ) const [inline],[virtual]`

Create a warmstart info describing that no change should be done.

This is really the task of a constructor, but BCP does not know the type of warmstart the user will use, so it will invoke this method for a warmstart that was created by the user. Tricky, isn't it?

Implements [BCP\\_warmstart](#).

Definition at line 73 of file `BCP_warmstart_basis.hpp`.

4.72.2.5 `virtual int BCP_warmstart_basis::storage_size ( ) const [inline],[virtual]`

Return how much memory it'll take to pack this warmstart info.

It is used when comparing which sort of storage is smaller.

Implements [BCP\\_warmstart](#).

Definition at line 79 of file `BCP_warmstart_basis.hpp`.

4.72.2.6 `virtual void BCP_warmstart_basis::update ( const BCP_warmstart *const change ) [virtual]`

Update the current data with the one in the argument.

If the argument is of explicit storage then just replace the current data. If it is relative and the current data is explicit then perform the update. Otherwise throw an exception.

Implements [BCP\\_warmstart](#).

4.72.2.7 `virtual BCP_warmstart* BCP_warmstart_basis::as_change ( const BCP_warmstart *const old_ws, const BCP_vec< int > &del_vars, const BCP_vec< int > &del_cuts, const double petol, const double detol ) const [virtual]`

Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.

However, if the currently stored data is shorter to store than the change, then this method can return a copy of the current data! The current data must be explicitly stored and `old_ws` must be either explicit or can contain no data. Otherwise an exception must be thrown.

## Parameters

<i>old_ws</i>	the old warmstart info
<i>del_vars</i>	the indices of the variables that are deleted from the formulation <code>old_ws</code> was created for
<i>del_cuts</i>	same for the cuts
<i>petol</i>	primal zero tolerance
<i>detol</i>	dual zero tolerance

Implements [BCP\\_warmstart](#).

4.72.2.8 `void BCP_warmstart_basis::pack ( BCP_buffer & buf ) const [inline]`

Pack the warmstart info into a buffer.

Definition at line 94 of file `BCP_warmstart_basis.hpp`.

The documentation for this class was generated from the following file:

- `BCP_warmstart_basis.hpp`

## 4.73 BCP\_warmstart\_dual Class Reference

This class describes a warmstart information that consists solely of the dual vector.

```
#include <BCP_warmstart_dual.hpp>
```

Inheritance diagram for `BCP_warmstart_dual`:

Collaboration diagram for `BCP_warmstart_dual`:

### Public Member Functions

- virtual **CoinWarmStart** \* [convert\\_to\\_CoinWarmStart](#) () const  
*Return an **CoinWarmStart** object that can be fed to the LP engine.*
- virtual `BCP_storage_t` [storage](#) () const  
*Return how the warmstarting info is stored.*
- virtual `BCP_warmstart` \* [clone](#) () const  
*Make a replica of the current warmstart information.*
- virtual `BCP_warmstart` \* [empty\\_wrt\\_this](#) () const  
*Create a warmstart info describing that no change should be done.*
- virtual `int` [storage\\_size](#) () const  
*Return how much memory it'll take to pack this warmstart info.*
- virtual `void` [update](#) (const `BCP_warmstart` \*const change)  
*Update the current data with the one in the argument.*
- virtual `BCP_warmstart` \* [as\\_change](#) (const `BCP_warmstart` \*const old\_ws, const `BCP_vec`< int > &del\_vars, const `BCP_vec`< int > &del\_cuts, const double petol, const double detol) const  
*Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.*
- `void` [pack](#) (`BCP_buffer` &buf) const  
*Pack the warmstart info into a buffer.*

### Constructors and destructor

- [BCP\\_warmstart\\_dual](#) ([BCP\\_buffer](#) &buf)  
*Create the object by unpacking it from a buffer.*
- [BCP\\_warmstart\\_dual](#) (const double \*first, const double \*last)  
*Create an explicitly stored warmstart info by considering the double array [first, last) as the dual vector.*
- [BCP\\_warmstart\\_dual](#) (const [BCP\\_warmstart\\_dual](#) &ws)  
*Copy constructor.*
- virtual [~BCP\\_warmstart\\_dual](#) ()

#### 4.73.1 Detailed Description

This class describes a warmstart information that consists solely of the dual vector.

Definition at line 20 of file [BCP\\_warmstart\\_dual.hpp](#).

#### 4.73.2 Constructor & Destructor Documentation

4.73.2.1 [BCP\\_warmstart\\_dual::BCP\\_warmstart\\_dual](#) ( const double \* first, const double \* last ) [inline]

Create an explicitly stored warmstart info by considering the double array [first, last) as the dual vector.

Definition at line 42 of file [BCP\\_warmstart\\_dual.hpp](#).

#### 4.73.3 Member Function Documentation

4.73.3.1 virtual [CoinWarmStart\\*](#) [BCP\\_warmstart\\_dual::convert\\_to\\_CoinWarmStart](#) ( ) const [virtual]

Return an **CoinWarmStart** object that can be fed to the LP engine.

The implementation for this class will return an **CoinWarmStartDual** object.

Implements [BCP\\_warmstart](#).

4.73.3.2 virtual [BCP\\_storage\\_t](#) [BCP\\_warmstart\\_dual::storage](#) ( ) const [inline],[virtual]

Return how the warmstarting info is stored.

Implements [BCP\\_warmstart](#).

Definition at line 56 of file [BCP\\_warmstart\\_dual.hpp](#).

4.73.3.3 virtual [BCP\\_warmstart\\*](#) [BCP\\_warmstart\\_dual::clone](#) ( ) const [inline],[virtual]

Make a replica of the current warmstart information.

Implements [BCP\\_warmstart](#).

Definition at line 58 of file [BCP\\_warmstart\\_dual.hpp](#).

4.73.3.4 virtual [BCP\\_warmstart\\*](#) [BCP\\_warmstart\\_dual::empty\\_wrt\\_this](#) ( ) const [inline],[virtual]

Create a warmstart info describing that no change should be done.

This is really the task of a constructor, but BCP does not know the type of warmstart the user will use, so it will invoke this method for a warmstart that was created by the user. Tricky, isn't it?

Implements [BCP\\_warmstart](#).

Definition at line 62 of file BCP\_warmstart\_dual.hpp.

**4.73.3.5** `virtual int BCP_warmstart_dual::storage_size ( ) const` `[inline],[virtual]`

Return how much memory it'll take to pack this warmstart info.

It is used when comparing which sort of storage is smaller.

Implements [BCP\\_warmstart](#).

Definition at line 68 of file BCP\_warmstart\_dual.hpp.

**4.73.3.6** `virtual void BCP_warmstart_dual::update ( const BCP_warmstart *const change )` `[virtual]`

Update the current data with the one in the argument.

If the argument is of explicit storage then just replace the current data. If it is relative and the current data is explicit then perform the update. Otherwise throw an exception.

Implements [BCP\\_warmstart](#).

**4.73.3.7** `virtual BCP_warmstart* BCP_warmstart_dual::as_change ( const BCP_warmstart *const old_ws, const BCP_vec< int > & del_vars, const BCP_vec< int > & del_cuts, const double petol, const double detol ) const` `[virtual]`

Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.

However, if the currently stored data is shorter to store than the change, then this method can return a copy of the current data! The current data must be explicitly stored and `old_ws` must be either explicit or can contain no data. Otherwise an exception must be thrown.

#### Parameters

<i>old_ws</i>	the old warmstart info
<i>del_vars</i>	the indices of the variables that are deleted from the formulation <code>old_ws</code> was created for
<i>del_cuts</i>	same for the cuts
<i>petol</i>	primal zero tolerance
<i>detol</i>	dual zero tolerance

Implements [BCP\\_warmstart](#).

**4.73.3.8** `void BCP_warmstart_dual::pack ( BCP_buffer & buf ) const` `[inline]`

Pack the warmstart info into a buffer.

Definition at line 83 of file BCP\_warmstart\_dual.hpp.

The documentation for this class was generated from the following file:

- BCP\_warmstart\_dual.hpp



## 4.74 BCP\_warmstart\_primaldual Class Reference

This class describes a warmstart information that consists solely of the dual vector.

```
#include <BCP_warmstart_primaldual.hpp>
```

Inheritance diagram for BCP\_warmstart\_primaldual:

Collaboration diagram for BCP\_warmstart\_primaldual:

### Public Member Functions

- virtual **CoinWarmStart** \* [convert\\_to\\_CoinWarmStart](#) () const  
*Return an **CoinWarmStart** object that can be fed to the LP engine.*
- virtual BCP\_storage\_t [storage](#) () const  
*Return how the warmstarting info is stored.*
- virtual BCP\_warmstart \* [clone](#) () const  
*Make a replica of the current warmstart information.*
- virtual BCP\_warmstart \* [empty\\_wrt\\_this](#) () const  
*Create a warmstart info describing that no change should be done.*
- virtual int [storage\\_size](#) () const  
*Return how much memory it'll take to pack this warmstart info.*
- virtual void [update](#) (const BCP\_warmstart \*const change)  
*Update the current data with the one in the argument.*
- virtual BCP\_warmstart \* [as\\_change](#) (const BCP\_warmstart \*const old\_ws, const BCP\_vec< int > &del\_vars, const BCP\_vec< int > &del\_cuts, const double petol, const double detol) const  
*Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.*
- void [pack](#) (BCP\_buffer &buf) const  
*Pack the warmstart info into a buffer.*

### Constructors and destructor

- BCP\_warmstart\_primaldual (BCP\_buffer &buf)  
*Create the object by unpacking it from a buffer.*
- BCP\_warmstart\_primaldual (const double \*pfirst, const double \*plast, const double \*dfirst, const double \*dlast)  
*Create an explicitly stored warmstart info by considering the double arrays `[ffirst,plast)` and `[dfirst,dlast)` as the primal and dual vectors.*
- BCP\_warmstart\_primaldual (const BCP\_warmstart\_primaldual &ws)  
*Copy constructor.*
- virtual ~BCP\_warmstart\_primaldual ()

#### 4.74.1 Detailed Description

This class describes a warmstart information that consists solely of the dual vector.

Definition at line 21 of file BCP\_warmstart\_primaldual.hpp.

### 4.74.2 Constructor & Destructor Documentation

4.74.2.1 `BCP_warmstart_primaldual::BCP_warmstart_primaldual ( const double * pfirst, const double * plast, const double * dfirst, const double * dlast ) [inline]`

Create an explicitly stored warmstart info by considering the double arrays [*fpfirst*,*plast*) and [*dfirst*,*dlast*) as the primal and dual vectors.

Definition at line 47 of file `BCP_warmstart_primaldual.hpp`.

### 4.74.3 Member Function Documentation

4.74.3.1 `virtual CoinWarmStart* BCP_warmstart_primaldual::convert_to_CoinWarmStart ( ) const [virtual]`

Return an **CoinWarmStart** object that can be fed to the LP engine.

The implementation for this class will return an **CoinWarmStartDual** object.

Implements [BCP\\_warmstart](#).

4.74.3.2 `virtual BCP_storage_t BCP_warmstart_primaldual::storage ( ) const [inline],[virtual]`

Return how the warmstarting info is stored.

Implements [BCP\\_warmstart](#).

Definition at line 63 of file `BCP_warmstart_primaldual.hpp`.

4.74.3.3 `virtual BCP_warmstart* BCP_warmstart_primaldual::clone ( ) const [inline],[virtual]`

Make a replica of the current warmstart information.

Implements [BCP\\_warmstart](#).

Definition at line 82 of file `BCP_warmstart_primaldual.hpp`.

4.74.3.4 `virtual BCP_warmstart* BCP_warmstart_primaldual::empty_wrt_this ( ) const [inline],[virtual]`

Create a warmstart info describing that no change should be done.

This is really the task of a constructor, but BCP does not know the type of warmstart the user will use, so it will invoke this method for a warmstart that was created by the user. Tricky, isn't it?

Implements [BCP\\_warmstart](#).

Definition at line 86 of file `BCP_warmstart_primaldual.hpp`.

4.74.3.5 `virtual int BCP_warmstart_primaldual::storage_size ( ) const [inline],[virtual]`

Return how much memory it'll take to pack this warmstart info.

It is used when comparing which sort of storage is smaller.

Implements [BCP\\_warmstart](#).

Definition at line 92 of file `BCP_warmstart_primaldual.hpp`.

4.74.3.6 `virtual void BCP_warmstart_primaldual::update ( const BCP_warmstart *const change ) [virtual]`

Update the current data with the one in the argument.

If the argument is of explicit storage then just replace the current data. If it is relative and the current data is explicit then perform the update. Otherwise throw an exception.

Implements [BCP\\_warmstart](#).

4.74.3.7 `virtual BCP_warmstart* BCP_warmstart_primaldual::as_change ( const BCP_warmstart *const old_ws, const BCP_vec< int > & del_vars, const BCP_vec< int > & del_cuts, const double petol, const double detol ) const [virtual]`

Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.

However, if the currently stored data is shorter to store than the change, then this method can return a copy of the current data! The current data must be explicitly stored and `old_ws` must be either explicit or can contain no data. Otherwise an exception must be thrown.

#### Parameters

<i>old_ws</i>	the old warmstart info
<i>del_vars</i>	the indices of the variables that are deleted from the formulation <code>old_ws</code> was created for
<i>del_cuts</i>	same for the cuts
<i>petol</i>	primal zero tolerance
<i>detol</i>	dual zero tolerance

Implements [BCP\\_warmstart](#).

4.74.3.8 `void BCP_warmstart_primaldual::pack ( BCP_buffer & buf ) const [inline]`

Pack the warmstart info into a buffer.

Definition at line 107 of file `BCP_warmstart_primaldual.hpp`.

The documentation for this class was generated from the following file:

- `BCP_warmstart_primaldual.hpp`

## 4.75 USER\_initialize Class Reference

This class is an abstract base class for the initializer class the user has to provide.

```
#include <BCP_USER.hpp>
```

### Public Member Functions

#### Destructor

- `virtual ~USER_initialize ()`  
*virtual destructor*

#### Message passing environment

- virtual `BCP_message_environment * msgenv_init (int argc, char *argv[])`  
*Create a message passing environment.*

### User object initialization in the processes

*These methods are invoked when the appropriate process starts.*

*They have to return pointers to user objects, i.e., objects derived from the return value of each init method. Those objects can be used by the user to store information about the problem; information that will be used in the member methods of the user objects.*

- virtual `BCP_tm_user * tm_init (BCP_tm_prob &p, const int argnum, const char *const *arglist)`
- virtual `BCP_ts_user * ts_init (BCP_ts_prob &p)`
- virtual `BCP_lp_user * lp_init (BCP_lp_prob &p)`
- virtual `BCP_vg_user * vg_init (BCP_vg_prob &p)`
- virtual `BCP_cg_user * cg_init (BCP_cg_prob &p)`
- virtual `BCP_vp_user * vp_init (BCP_vp_prob &p)`
- virtual `BCP_cp_user * cp_init (BCP_cp_prob &p)`
- virtual `BCP_user_pack * packer_init (BCP_user_class *p)`

## 4.75.1 Detailed Description

This class is an abstract base class for the initializer class the user has to provide.

The user will have to return an instance of the initializer class when the `BCP_user_init()` function is invoked. The member methods of that instance will be invoked to create the various objects (well, pointers to them) that are used/controlled by the user during the course of a run.

Definition at line 160 of file `BCP_USER.hpp`.

## 4.75.2 Member Function Documentation

### 4.75.2.1 virtual `BCP_message_environment* USER_initialize::msgenv_init ( int argc, char * argv[] )` [virtual]

Create a message passing environment.

Currently implemented environments are single and PVM, the default is single. To use PVM, the user has to override this method and return a pointer to a new `BCP_pvm_environment` object.

The documentation for this class was generated from the following file:

- `BCP_USER.hpp`

File Documentation

# Index

- [\\_LowerBound](#)
    - [BCP\\_col, 37](#)
    - [BCP\\_row, 116](#)
  - [\\_Objective](#)
    - [BCP\\_col, 37](#)
  - [\\_UpperBound](#)
    - [BCP\\_col, 38](#)
    - [BCP\\_row, 116](#)
  - [\\_data](#)
    - [BCP\\_buffer, 27](#)
  - [\\_delete\\_vars](#)
    - [BCP\\_solution\\_generic, 124](#)
  - [\\_lb](#)
    - [BCP\\_cut, 43](#)
    - [BCP\\_var, 149](#)
  - [\\_max\\_size](#)
    - [BCP\\_buffer, 27](#)
  - [\\_msgtag](#)
    - [BCP\\_buffer, 26](#)
  - [\\_obj](#)
    - [BCP\\_var, 149](#)
  - [\\_objective](#)
    - [BCP\\_solution\\_generic, 124](#)
  - [\\_pos](#)
    - [BCP\\_buffer, 26](#)
  - [\\_sender](#)
    - [BCP\\_buffer, 26](#)
  - [\\_size](#)
    - [BCP\\_buffer, 27](#)
  - [\\_storage](#)
    - [BCP\\_problem\\_core\\_change, 111](#)
  - [\\_ub](#)
    - [BCP\\_cut, 43](#)
    - [BCP\\_var, 149](#)
  - [\\_values](#)
    - [BCP\\_solution\\_generic, 125](#)
  - [\\_var\\_type](#)
    - [BCP\\_var, 149](#)
  - [\\_vars](#)
    - [BCP\\_solution\\_generic, 125](#)
- [~BCP\\_buffer](#)
  - [BCP\\_buffer, 23](#)
- [~BCP\\_cg\\_prob](#)
  - [BCP\\_cg\\_prob, 29](#)
- [~BCP\\_cg\\_user](#)
  - [BCP\\_cg\\_user, 33](#)
- [~BCP\\_col](#)
  - [BCP\\_col, 36](#)
- [~BCP\\_cut](#)
  - [BCP\\_cut, 40](#)
- [~BCP\\_cut\\_algo](#)
  - [BCP\\_cut\\_algo, 44](#)
- [~BCP\\_cut\\_core](#)
  - [BCP\\_cut\\_core, 45](#)
- [~BCP\\_cut\\_set](#)
  - [BCP\\_cut\\_set, 46](#)
- [~BCP\\_fatal\\_error](#)
  - [BCP\\_fatal\\_error, 48](#)
- [~BCP\\_internal\\_brobj](#)
  - [BCP\\_internal\\_brobj, 49](#)
- [~BCP\\_lp\\_branching\\_object](#)
  - [BCP\\_lp\\_branching\\_object, 53](#)
- [~BCP\\_lp\\_result](#)
  - [BCP\\_lp\\_result, 73](#)
- [~BCP\\_lp\\_user](#)
  - [BCP\\_lp\\_user, 80](#)
- [~BCP\\_message\\_environment](#)
  - [BCP\\_message\\_environment, 93](#)
- [~BCP\\_parameter](#)
  - [BCP\\_parameter, 98](#)
- [~BCP\\_parameter\\_set](#)
  - [BCP\\_parameter\\_set, 100](#)
- [~BCP\\_presolved\\_lp\\_brobj](#)
  - [BCP\\_presolved\\_lp\\_brobj, 103](#)
- [~BCP\\_problem\\_core](#)
  - [BCP\\_problem\\_core, 106](#)
- [~BCP\\_problem\\_core\\_change](#)
  - [BCP\\_problem\\_core\\_change, 110](#)
- [~BCP\\_row](#)
  - [BCP\\_row, 114](#)
- [~BCP\\_solution](#)
  - [BCP\\_solution, 122](#)
- [~BCP\\_solution\\_generic](#)
  - [BCP\\_solution\\_generic, 124](#)
- [~BCP\\_tm\\_user](#)
  - [BCP\\_tm\\_user, 137](#)
- [~BCP\\_ts\\_user](#)
  - [BCP\\_ts\\_user, 142](#)
- [~BCP\\_var](#)
  - [BCP\\_var, 146](#)

- ~BCP\_var\_algo
  - BCP\_var\_algo, 150
- ~BCP\_var\_core
  - BCP\_var\_core, 152
- ~BCP\_var\_set
  - BCP\_var\_set, 153
- ~BCP\_vec
  - BCP\_vec, 158
- ~BCP\_vec\_change
  - BCP\_vec\_change, 167
- ~BCP\_vg\_prob
  - BCP\_vg\_prob, 170
- ~BCP\_vg\_user
  - BCP\_vg\_user, 174
- ~BCP\_warmstart
  - BCP\_warmstart, 176
- action
  - BCP\_presolved\_lp\_brojb, 104
- add
  - BCP\_lp\_statistics, 75
- add\_entry
  - BCP\_solution\_generic, 124
- add\_free\_ids
  - BCP\_scheduler, 117
- affected\_cutnum
  - BCP\_internal\_brojb, 50
- affected\_varnum
  - BCP\_internal\_brojb, 50
- alive
  - BCP\_message\_environment, 94
  - BCP\_single\_environment, 119
- allocate
  - BCP\_vec, 158
- append
  - BCP\_cut\_set, 47
  - BCP\_var\_set, 153
  - BCP\_vec, 161
- append\_branching\_vars
  - BCP\_lp\_user, 90
- apply\_child\_bd
  - BCP\_lp\_branching\_object, 55
- apply\_child\_bounds
  - BCP\_internal\_brojb, 50
- as\_change
  - BCP\_warmstart, 176
  - BCP\_warmstart\_basis, 179
  - BCP\_warmstart\_dual, 182
  - BCP\_warmstart\_primaldual, 185
- assign
  - BCP\_col, 37
  - BCP\_row, 115
  - BCP\_vec, 161
- BCP\_MemPool, 92
- BCP\_buffer, 21
  - \_data, 27
  - \_max\_size, 27
  - \_msgtag, 26
  - \_pos, 26
  - \_sender, 26
  - \_size, 27
  - ~BCP\_buffer, 23
  - BCP\_buffer, 23
  - clear, 25
  - data, 24
  - make\_fit, 24
  - msgtag, 24
  - operator=, 24
  - pack, 25, 26
  - sender, 24
  - set\_content, 24
  - set\_position, 24
  - set\_size, 24
  - size, 24
  - unpack, 25, 26
- BCP\_cg\_par, 27
- BCP\_cg\_prob, 27
  - ~BCP\_cg\_prob, 29
  - BCP\_cg\_prob, 29
  - core, 30
  - has\_ub, 29
  - msg\_buf, 30
  - msg\_env, 30
  - node\_index, 31
  - node\_iteration, 31
  - node\_level, 31
  - packer, 30
  - par, 30
  - phase, 31
  - probe\_messages, 29
  - sender, 31
  - ub, 29
  - unpack\_var, 29
  - upper\_bound, 30
  - user, 29
  - vars, 30
  - x, 30
- BCP\_cg\_user, 31
  - ~BCP\_cg\_user, 33
  - generate\_cuts, 34
  - send\_cut, 33
  - unpack\_module\_data, 33
  - unpack\_primal\_solution, 33
- BCP\_col, 34
  - \_LowerBound, 37
  - \_Objective, 37
  - \_UpperBound, 38
  - ~BCP\_col, 36

- assign, [37](#)
- BCP\_col, [35](#), [36](#)
- copy, [37](#)
- LowerBound, [36](#)
- Objective, [36](#)
- operator=, [37](#)
- UpperBound, [36](#), [37](#)
- BCP\_cut, [38](#)
  - \_lb, [43](#)
  - \_ub, [43](#)
  - ~BCP\_cut, [40](#)
  - BCP\_cut, [40](#)
  - bcpind, [40](#)
  - change\_bounds, [42](#)
  - change\_lb\_ub\_st, [42](#)
  - decrease\_effective\_count, [41](#)
  - dont\_send\_to\_pool, [40](#), [42](#)
  - effective\_count, [40](#)
  - increase\_effective\_count, [41](#)
  - is\_non\_removable, [41](#)
  - is\_to\_be\_removed, [41](#)
  - lb, [40](#)
  - make\_active, [42](#)
  - make\_non\_removable, [42](#)
  - make\_to\_be\_removed, [42](#)
  - obj\_type, [40](#)
  - set\_bcpind, [42](#)
  - set\_effective\_count, [41](#)
  - set\_lb, [41](#)
  - set\_status, [42](#)
  - set\_ub, [41](#)
  - status, [40](#)
  - ub, [40](#)
- BCP\_cut\_algo, [43](#)
  - ~BCP\_cut\_algo, [44](#)
  - BCP\_cut\_algo, [44](#)
  - obj\_type, [44](#)
- BCP\_cut\_core, [44](#)
  - ~BCP\_cut\_core, [45](#)
  - BCP\_cut\_core, [45](#)
  - obj\_type, [45](#)
- BCP\_cut\_set, [45](#)
  - ~BCP\_cut\_set, [46](#)
  - append, [47](#)
  - BCP\_cut\_set, [46](#)
  - move\_deletable\_to\_pool, [47](#)
  - set\_lb\_ub, [47](#)
  - set\_lb\_ub\_st, [47](#)
- BCP\_fatal\_error, [47](#)
  - ~BCP\_fatal\_error, [48](#)
  - BCP\_fatal\_error, [48](#)
- BCP\_internal\_brobj, [48](#)
  - ~BCP\_internal\_brobj, [49](#)
  - affected\_cutnum, [50](#)
  - affected\_varnum, [50](#)
  - apply\_child\_bounds, [50](#)
  - BCP\_internal\_brobj, [49](#)
  - child\_num, [50](#)
  - cut\_bounds\_child, [50](#)
  - cut\_positions, [50](#)
  - pack, [51](#)
  - unpack, [51](#)
  - var\_bounds\_child, [50](#)
  - var\_positions, [50](#)
- BCP\_lp\_branching\_object, [51](#)
  - ~BCP\_lp\_branching\_object, [53](#)
  - apply\_child\_bd, [55](#)
  - BCP\_lp\_branching\_object, [53](#)
  - child\_num, [55](#)
  - cuts\_added, [54](#)
  - cuts\_affected, [53](#)
  - cuts\_to\_add, [55](#)
  - forced\_cut\_bd, [56](#)
  - forced\_cut\_bd\_child, [54](#)
  - forced\_cut\_pos, [55](#)
  - forced\_var\_bd, [56](#)
  - forced\_var\_bd\_child, [54](#)
  - forced\_var\_pos, [55](#)
  - implied\_cut\_bd\_child, [54](#)
  - implied\_var\_bd\_child, [54](#)
  - init\_pos\_for\_added, [55](#)
  - print\_branching\_info, [55](#)
  - vars\_added, [54](#)
  - vars\_affected, [53](#)
  - vars\_to\_add, [55](#)
- BCP\_lp\_cut\_pool, [56](#)
- BCP\_lp\_integer\_branching\_object, [56](#)
- BCP\_lp\_node, [57](#)
  - user\_data, [58](#)
- BCP\_lp\_par, [58](#)
  - BranchOnCuts, [59](#)
  - BranchingObjectComparison, [62](#)
  - ChildPreference, [62](#)
  - chr\_params, [59](#)
  - CompareNewCutsToOldOnes, [59](#)
  - CompareNewVarsToOldOnes, [59](#)
  - CutEffectiveCountBeforePool, [61](#)
  - CutPoolCheckFrequency, [61](#)
  - CutViolationNorm, [62](#)
  - dbl\_params, [63](#)
  - DeletedColToCompress\_Frac, [63](#)
  - DeletedColToCompress\_Min, [62](#)
  - DeletedRowToCompress\_Frac, [63](#)
  - DeletedRowToCompress\_Min, [62](#)
  - DoReducedCostFixingAtAnything, [60](#)
  - DoReducedCostFixingAtZero, [59](#)
  - FeasibilityTest, [62](#)
  - FirstLP\_AllCutsTimeout, [64](#)

FirstLP\_AllVarsTimeout, 64  
 FirstLP\_FirstCutTimeout, 63  
 FirstLP\_FirstVarTimeout, 64  
 Granularity, 63  
 IneffectiveBeforeDelete, 62  
 IneffectiveConstraints, 61  
 InfoForCG, 63  
 InfoForVG, 63  
 int\_params, 61  
 IntegerTolerance, 63  
 LaterLP\_AllCutsTimeout, 64  
 LaterLP\_AllVarsTimeout, 64  
 LaterLP\_FirstCutTimeout, 63  
 LaterLP\_FirstVarTimeout, 64  
 LogFileName, 64  
 LpVerb\_AddedCutCount, 60  
 LpVerb\_AddedVarCount, 60  
 LpVerb\_ChildrenInfo, 60  
 LpVerb\_ColumnGenerationInfo, 60  
 LpVerb\_CutsToCutPoolCount, 60  
 LpVerb\_FathomInfo, 60  
 LpVerb\_FinalRelaxedSolution, 60  
 LpVerb\_First, 60  
 LpVerb\_GeneratedCutCount, 61  
 LpVerb\_GeneratedVarCount, 61  
 LpVerb\_IterationCount, 60  
 LpVerb\_Last, 61  
 LpVerb\_LpMatrixSize, 60  
 LpVerb\_LpSolutionValue, 60  
 LpVerb\_MatrixCompression, 60  
 LpVerb\_NodeTime, 60  
 LpVerb\_PresolvePositions, 60  
 LpVerb\_PresolveResult, 60  
 LpVerb\_ProcessedNodeIndex, 60  
 LpVerb\_RelaxedSolution, 60  
 LpVerb\_ReportCutGenTimeout, 60  
 LpVerb\_ReportLocalCutPoolSize, 61  
 LpVerb\_ReportLocalVarPoolSize, 61  
 LpVerb\_ReportVarGenTimeout, 61  
 LpVerb\_RepricingResult, 61  
 LpVerb\_RowEffectivenessCount, 61  
 LpVerb\_StrongBranchPositions, 61  
 LpVerb\_StrongBranchResult, 61  
 LpVerb\_VarTightening, 61  
 LpVerb\_VarsToVarPoolCount, 60  
 MaxCutsAddedPerIteration, 62  
 MaxLeftoverCutFrac, 63  
 MaxLeftoverCutNum, 62  
 MaxNonDualFeasToAdd\_Frac, 63  
 MaxNonDualFeasToAdd\_Max, 62  
 MaxNonDualFeasToAdd\_Min, 62  
 MaxPresolverIter, 62  
 MaxRunTime, 64  
 MaxVarsAddedPerIteration, 62  
 MessagePassingIsSerial, 60  
 NiceLevel, 61  
 NoCompressionAtFathom, 60  
 ReportWhenDefaultIsExecuted, 60  
 ScaleMatrix, 61  
 SendFathomedNodeDesc, 60  
 SlackCutDiscardingStrategy, 61  
 str\_array\_params, 64  
 str\_params, 64  
 StrongBranch\_CloseToHalfNum, 62  
 StrongBranchNum, 62  
 VarPoolCheckFrequency, 61  
 WarmstartInfo, 62  
 BCP\_lp\_parent, 65  
 BCP\_lp\_prob, 65  
   intAndSosObjects, 67  
   node, 67  
   packer, 67  
   parent, 68  
   warmstartRoot, 68  
 BCP\_lp\_relax, 68  
   BCP\_lp\_relax, 70, 71  
   clear, 72  
   ColLowerBound, 71  
   ColUpperBound, 71  
   colnum, 71  
   erase\_col\_set, 72  
   erase\_row\_set, 72  
   Objective, 71  
   operator=, 72  
   pack, 72  
   reserve, 72  
   RowLowerBound, 71  
   RowUpperBound, 71  
   rownum, 71  
   unpack, 72  
 BCP\_lp\_result, 72  
   ~BCP\_lp\_result, 73  
   BCP\_lp\_result, 73  
   dualTolerance, 74  
   fake\_objective\_value, 74  
   get\_results, 74  
   primalTolerance, 74  
 BCP\_lp\_sos\_branching\_object, 74  
 BCP\_lp\_statistics, 75  
   add, 75  
 BCP\_lp\_user, 75  
   ~BCP\_lp\_user, 80  
   append\_branching\_vars, 90  
   branch\_close\_to\_half, 89  
   branch\_close\_to\_one, 89  
   compare\_branching\_candidates, 90  
   compare\_cuts, 88  
   compare\_vars, 88



- compute\_lower\_bound, 83
- cuts\_to\_rows, 86
- display\_lp\_solution, 85
- generate\_cuts\_in\_lp, 87
- generate\_heuristic\_solution, 84
- generate\_vars\_in\_lp, 87
- initialize\_int\_and\_sos\_list, 82
- initialize\_new\_search\_tree\_node, 82
- initialize\_solver\_interface, 81
- load\_problem, 82
- logical\_fixing, 88
- modify\_lp\_parameters, 82
- over\_ub, 81
- pack\_dual\_solution, 85
- pack\_feasible\_solution, 84
- pack\_primal\_solution, 85
- process\_lp\_result, 83
- process\_message, 81
- purge\_slack\_pool, 90
- reduced\_cost\_fixing, 89
- restore\_feasibility, 86
- select\_branching\_candidates, 89
- select\_fractions, 81
- select\_nonzeros, 81
- select\_positives, 81
- select\_zeros, 81
- set\_actions\_for\_children, 90
- set\_user\_data\_for\_children, 90
- test\_binary, 84
- test\_feasibility, 83
- test\_full, 84
- test\_integral, 84
- unpack\_module\_data, 81
- vars\_to\_cols, 86
- BCP\_lp\_var\_pool, 91
- BCP\_lp\_waiting\_col, 91
- BCP\_lp\_waiting\_row, 91
- BCP\_message\_environment, 92
  - ~BCP\_message\_environment, 93
  - alive, 94
  - multicast, 94
  - num\_procs, 96
  - parent\_process, 94
  - probe, 95
  - receive, 95
  - register\_process, 94
  - send, 94
  - start\_process, 95
  - start\_processes, 95, 96
- BCP\_node\_change, 96
- BCP\_node\_storage\_in\_tm, 96
- BCP\_obj\_change, 97
- BCP\_obj\_set\_change, 97
- BCP\_parameter, 97
  - ~BCP\_parameter, 98
  - BCP\_parameter, 98
  - index, 98
  - type, 98
- BCP\_parameter\_set
  - ~BCP\_parameter\_set, 100
  - BCP\_parameter\_set, 100
  - pack, 101
  - read\_from\_arglist, 101
  - read\_from\_file, 101
  - read\_from\_stream, 101
  - unpack, 101
  - write\_to\_stream, 101
- BCP\_parameter\_set< Par >, 99
- BCP\_presolved\_lp\_brobj, 102
  - ~BCP\_presolved\_lp\_brobj, 103
  - action, 104
  - BCP\_presolved\_lp\_brobj, 103
  - candidate, 103
  - fake\_objective\_values, 105
  - fathomable, 104
  - get\_lower\_bounds, 104
  - get\_results, 105
  - had\_numerical\_problems, 104
  - lpres, 103
  - set\_lower\_bounds, 104
  - set\_objective\_values, 105
  - user\_data, 104
- BCP\_problem\_core, 105
  - ~BCP\_problem\_core, 106
  - BCP\_problem\_core, 106
  - cutnum, 107
  - cuts, 107
  - matrix, 107
  - pack, 107
  - unpack, 107
  - varnum, 107
  - vars, 107
- BCP\_problem\_core\_change, 108
  - \_storage, 111
  - ~BCP\_problem\_core\_change, 110
  - BCP\_problem\_core\_change, 109, 110
  - cut\_ch, 112
  - cut\_pos, 112
  - cutnum, 110
  - ensure\_explicit, 110
  - make\_wrtcore\_if\_shorter, 111
  - operator=, 110
  - pack, 111
  - pack\_size, 111
  - storage, 110
  - swap, 111
  - unpack, 111
  - update, 111

- var\_ch, 112
- var\_pos, 112
- varnum, 110
- BCP\_process, 112
- BCP\_row, 113
  - \_LowerBound, 116
  - \_UpperBound, 116
  - ~BCP\_row, 114
  - assign, 115
  - BCP\_row, 114
  - copy, 115
  - LowerBound, 115
  - operator=, 115
  - UpperBound, 115
- BCP\_scheduler, 116
  - add\_free\_ids, 117
  - BCP\_scheduler, 117
  - release\_sb\_id, 118
  - request\_node\_id, 118
  - request\_sb\_ids, 117
  - setParams, 117
- BCP\_single\_environment, 118
  - alive, 119
  - multicast, 120
  - parent\_process, 119
  - probe, 120
  - receive, 120
  - register\_process, 119
  - send, 120
  - start\_process, 120, 121
  - start\_processes, 121
- BCP\_slave\_params, 121
- BCP\_solution, 122
  - ~BCP\_solution, 122
  - objective\_value, 122
- BCP\_solution\_generic, 123
  - \_delete\_vars, 124
  - \_objective, 124
  - \_values, 125
  - \_vars, 125
  - ~BCP\_solution\_generic, 124
  - add\_entry, 124
  - BCP\_solution\_generic, 124
  - display, 124
  - objective\_value, 124
  - set\_objective\_value, 124
- BCP\_string, 125
- BCP\_tm\_flags, 125
  - root\_pricing\_unpacked, 126
- BCP\_tm\_node, 126
- BCP\_tm\_node\_data, 127
- BCP\_tm\_node\_to\_send, 127
  - receive\_cuts, 128
  - receive\_node\_desc, 128
  - receive\_vars, 128
- BCP\_tm\_par, 128
  - CgMachines, 132
  - CgProcessNum, 131
  - chr\_params, 129
  - CpMachines, 132
  - CpProcessNum, 131
  - dbl\_params, 131
  - DebugCgProcesses, 129
  - DebugCpProcesses, 130
  - DebugLpProcesses, 129
  - DebugVgProcesses, 129
  - DebugVpProcesses, 130
  - ExecutableName, 132
  - GenerateVars, 130
  - Granularity, 131
  - int\_params, 130
  - LPscheduler\_MaxNodeIDNum, 131
  - LPscheduler\_MaxSbIDNum, 131
  - LPscheduler\_MinSbIDNum, 131
  - LPscheduler\_OverEstimationStatic, 131
  - LogFileName, 132
  - LpMachines, 132
  - LpProcessNum, 131
  - MaxHeapSize, 131
  - MaxRunTime, 131
  - MessagePassingIsSerial, 130
  - NiceLevel, 131
  - QualityRatioToAllowDiving\_HasUB, 131
  - QualityRatioToAllowDiving\_NoUB, 131
  - ReadRootCutsFrom, 132
  - RemoveExploredBranches, 130
  - ReportWhenDefaultIsExecuted, 130
  - SaveRootCutsTo, 132
  - str\_array\_params, 132
  - str\_params, 131
  - TerminationGap\_Absolute, 131
  - TerminationGap\_Relative, 131
  - TmTimeout, 131
  - TmVerb\_AllFeasibleSolution, 130
  - TmVerb\_AllFeasibleSolutionValue, 130
  - TmVerb\_BestFeasibleSolution, 130
  - TmVerb\_BetterFeasibleSolution, 130
  - TmVerb\_BetterFeasibleSolutionValue, 130
  - TmVerb\_FinalStatistics, 130
  - TmVerb\_First, 130
  - TmVerb\_Last, 130
  - TmVerb\_NewPhaseStart, 130
  - TmVerb\_PrunedNodeInfo, 130
  - TmVerb\_ReportDefault, 130
  - TmVerb\_SingleLineInfoFrequency, 131
  - TmVerb\_TimeOfImprovingSolution, 130
  - TmVerb\_TrimmedNum, 130
  - TreeSearchStrategy, 131

- TrimTreeBeforeNewPhase, 130
- UnconditionalDiveProbability, 131
- UpperBound, 131
- VerbosityShutUp, 130
- VgMachines, 132
- VgProcessNum, 131
- VpMachines, 132
- VpProcessNum, 131
- WarmstartInfo, 131
- BCP\_tm\_prob, 132
  - lb\_multiplier, 134
  - packer, 134
  - root\_node\_sent\_, 134
- BCP\_tm\_stat, 135
- BCP\_tm\_user, 135
  - ~BCP\_tm\_user, 137
  - create\_root, 138
  - display\_final\_information, 138
  - display\_node\_information, 138
  - init\_new\_phase, 139
  - initialize\_core, 138
  - lower\_bound, 137
  - pack\_module\_data, 137
  - process\_message, 138
  - replace\_solution, 138
  - unpack\_feasible\_solution, 137
- BCP\_tree, 139
- BCP\_ts\_node\_data, 140
- BCP\_ts\_par, 140
  - int\_params, 140
  - MaxHeapSize, 140
- BCP\_ts\_prob, 140
- BCP\_ts\_user, 141
  - ~BCP\_ts\_user, 142
  - unpack\_module\_data, 142
- BCP\_user\_class, 142
- BCP\_user\_data, 142
- BCP\_user\_pack, 143
  - user\_class, 143
- BCP\_var, 144
  - \_lb, 149
  - \_obj, 149
  - \_ub, 149
  - \_var\_type, 149
  - ~BCP\_var, 146
  - BCP\_var, 146
  - bcpind, 147
  - change\_bounds, 148
  - display, 149
  - dont\_send\_to\_pool, 147, 148
  - is\_fixed, 147
  - is\_fixed\_to\_zero, 147
  - is\_non\_removable, 147
  - is\_removable, 147
  - is\_to\_be\_removed, 147
  - lb, 146
  - make\_active, 149
  - make\_non\_removable, 149
  - make\_to\_be\_removed, 149
  - obj, 146
  - obj\_type, 146
  - set\_bcpind, 148
  - set\_lb, 148
  - set\_lb\_ub, 148
  - set\_obj, 148
  - set\_status, 148
  - set\_ub, 148
  - set\_var\_type, 148
  - status, 147
  - ub, 146
  - var\_type, 146
- BCP\_var\_algo, 150
  - ~BCP\_var\_algo, 150
  - BCP\_var\_algo, 150
  - obj\_type, 151
- BCP\_var\_core, 151
  - ~BCP\_var\_core, 152
  - BCP\_var\_core, 152
  - obj\_type, 152
- BCP\_var\_set, 152
  - ~BCP\_var\_set, 153
  - append, 153
  - deletable, 154
  - set\_lb\_ub, 153
  - set\_lb\_ub\_st, 154
- BCP\_vec
  - ~BCP\_vec, 158
  - allocate, 158
  - append, 161
  - assign, 161
  - BCP\_vec, 157, 158
  - back, 160
  - begin, 158, 159
  - capacity, 159
  - clear, 162
  - deallocate, 158
  - empty, 159
  - end, 159
  - end\_of\_storage, 164
  - entry, 159
  - erase, 163
  - erase\_by\_index, 163
  - finish, 164
  - front, 160
  - index, 159
  - insert, 161
  - insert\_aux, 158
  - keep, 162

- keep\_by\_index, 162, 163
- operator=, 160
- operator[], 160
- pop\_back, 162
- push\_back, 161
- reserve, 160
- size, 159
- start, 164
- swap, 160
- unchecked\_erase\_by\_index, 163
- unchecked\_keep\_by\_index, 163
- unchecked\_push\_back, 161
- unchecked\_update, 162
- update, 162
- BCP\_vec< T >, 154
- BCP\_vec\_change
  - ~BCP\_vec\_change, 167
  - BCP\_vec\_change, 165, 167
  - pack, 167
  - storage\_size, 167
  - unpack, 168
  - update, 167
- BCP\_vec\_change< T >, 164
- BCP\_vg\_par, 168
  - chr\_params, 168
  - int\_params, 168
  - MessagePassingIsSerial, 168
  - NiceLevel, 168
  - ReportWhenDefaultIsExecuted, 168
- BCP\_vg\_prob, 169
  - ~BCP\_vg\_prob, 170
  - BCP\_vg\_prob, 170
  - core, 171
  - cuts, 172
  - has\_ub, 170
  - msg\_buf, 171
  - msg\_env, 171
  - node\_index, 172
  - node\_iteration, 172
  - node\_level, 172
  - packer, 171
  - par, 171
  - phase, 172
  - pi, 172
  - probe\_messages, 171
  - sender, 172
  - ub, 170
  - unpack\_cut, 171
  - upper\_bound, 172
  - user, 171
- BCP\_vg\_user, 173
  - ~BCP\_vg\_user, 174
  - generate\_vars, 175
  - send\_var, 175
  - unpack\_dual\_solution, 175
  - unpack\_module\_data, 175
- BCP\_warmstart, 175
  - ~BCP\_warmstart, 176
  - as\_change, 176
  - clone, 177
  - empty\_wrt\_this, 177
  - storage, 176
  - storage\_size, 177
  - update, 176
- BCP\_warmstart\_basis, 177
  - as\_change, 179
  - clone, 179
  - convert\_to\_CoinWarmStart, 178
  - empty\_wrt\_this, 179
  - pack, 180
  - storage, 178
  - storage\_size, 179
  - update, 179
- BCP\_warmstart\_dual, 180
  - as\_change, 182
  - BCP\_warmstart\_dual, 181
  - clone, 181
  - convert\_to\_CoinWarmStart, 181
  - empty\_wrt\_this, 181
  - pack, 182
  - storage, 181
  - storage\_size, 182
  - update, 182
- BCP\_warmstart\_primaldual, 183
  - as\_change, 185
  - BCP\_warmstart\_primaldual, 184
  - clone, 184
  - convert\_to\_CoinWarmStart, 184
  - empty\_wrt\_this, 184
  - pack, 185
  - storage, 184
  - storage\_size, 184
  - update, 184
- back
  - BCP\_vec, 160
- bcpind
  - BCP\_cut, 40
  - BCP\_var, 147
- begin
  - BCP\_vec, 158, 159
- branch\_close\_to\_half
  - BCP\_lp\_user, 89
- branch\_close\_to\_one
  - BCP\_lp\_user, 89
- BranchOnCuts
  - BCP\_lp\_par, 59
- BranchingObjectComparison
  - BCP\_lp\_par, 62

- candidate
  - BCP\_presolved\_lp\_brobj, 103
- capacity
  - BCP\_vec, 159
- CgMachines
  - BCP\_tm\_par, 132
- CgProcessNum
  - BCP\_tm\_par, 131
- change\_bounds
  - BCP\_cut, 42
  - BCP\_var, 148
- change\_lb\_ub\_st
  - BCP\_cut, 42
- child\_num
  - BCP\_internal\_brobj, 50
  - BCP\_lp\_branching\_object, 55
- ChildPreference
  - BCP\_lp\_par, 62
- chr\_params
  - BCP\_lp\_par, 59
  - BCP\_tm\_par, 129
  - BCP\_vg\_par, 168
- clear
  - BCP\_buffer, 25
  - BCP\_lp\_relax, 72
  - BCP\_vec, 162
- clone
  - BCP\_warmstart, 177
  - BCP\_warmstart\_basis, 179
  - BCP\_warmstart\_dual, 181
  - BCP\_warmstart\_primaldual, 184
- ColLowerBound
  - BCP\_lp\_relax, 71
- ColUpperBound
  - BCP\_lp\_relax, 71
- colnum
  - BCP\_lp\_relax, 71
- compare\_branching\_candidates
  - BCP\_lp\_user, 90
- compare\_cuts
  - BCP\_lp\_user, 88
- compare\_vars
  - BCP\_lp\_user, 88
- CompareNewCutsToOldOnes
  - BCP\_lp\_par, 59
- CompareNewVarsToOldOnes
  - BCP\_lp\_par, 59
- compute\_lower\_bound
  - BCP\_lp\_user, 83
- convert\_to\_CoinWarmStart
  - BCP\_warmstart\_basis, 178
  - BCP\_warmstart\_dual, 181
  - BCP\_warmstart\_primaldual, 184
- copy
  - BCP\_col, 37
  - BCP\_row, 115
- core
  - BCP\_cg\_prob, 30
  - BCP\_vg\_prob, 171
- CpMachines
  - BCP\_tm\_par, 132
- CpProcessNum
  - BCP\_tm\_par, 131
- create\_root
  - BCP\_tm\_user, 138
- cut\_bounds\_child
  - BCP\_internal\_brobj, 50
- cut\_ch
  - BCP\_problem\_core\_change, 112
- cut\_pos
  - BCP\_problem\_core\_change, 112
- cut\_positions
  - BCP\_internal\_brobj, 50
- CutEffectiveCountBeforePool
  - BCP\_lp\_par, 61
- CutPoolCheckFrequency
  - BCP\_lp\_par, 61
- CutViolationNorm
  - BCP\_lp\_par, 62
- cutnum
  - BCP\_problem\_core, 107
  - BCP\_problem\_core\_change, 110
- cuts
  - BCP\_problem\_core, 107
  - BCP\_vg\_prob, 172
- cuts\_added
  - BCP\_lp\_branching\_object, 54
- cuts\_affected
  - BCP\_lp\_branching\_object, 53
- cuts\_to\_add
  - BCP\_lp\_branching\_object, 55
- cuts\_to\_rows
  - BCP\_lp\_user, 86
- data
  - BCP\_buffer, 24
- dbl\_params
  - BCP\_lp\_par, 63
  - BCP\_tm\_par, 131
- deallocate
  - BCP\_vec, 158
- DebugCgProcesses
  - BCP\_tm\_par, 129
- DebugCpProcesses
  - BCP\_tm\_par, 130
- DebugLpProcesses
  - BCP\_tm\_par, 129
- DebugVgProcesses

- BCP\_tm\_par, [129](#)
- DebugVpProcesses
  - BCP\_tm\_par, [130](#)
- decrease\_effective\_count
  - BCP\_cut, [41](#)
- deletable
  - BCP\_var\_set, [154](#)
- DeletedColToCompress\_Frac
  - BCP\_lp\_par, [63](#)
- DeletedColToCompress\_Min
  - BCP\_lp\_par, [62](#)
- DeletedRowToCompress\_Frac
  - BCP\_lp\_par, [63](#)
- DeletedRowToCompress\_Min
  - BCP\_lp\_par, [62](#)
- display
  - BCP\_solution\_generic, [124](#)
  - BCP\_var, [149](#)
- display\_final\_information
  - BCP\_tm\_user, [138](#)
- display\_lp\_solution
  - BCP\_lp\_user, [85](#)
- display\_node\_information
  - BCP\_tm\_user, [138](#)
- DoReducedCostFixingAtAnything
  - BCP\_lp\_par, [60](#)
- DoReducedCostFixingAtZero
  - BCP\_lp\_par, [59](#)
- dont\_send\_to\_pool
  - BCP\_cut, [40](#), [42](#)
  - BCP\_var, [147](#), [148](#)
- dualTolerance
  - BCP\_lp\_result, [74](#)
- effective\_count
  - BCP\_cut, [40](#)
- empty
  - BCP\_vec, [159](#)
- empty\_wrt\_this
  - BCP\_warmstart, [177](#)
  - BCP\_warmstart\_basis, [179](#)
  - BCP\_warmstart\_dual, [181](#)
  - BCP\_warmstart\_primaldual, [184](#)
- end
  - BCP\_vec, [159](#)
- end\_of\_storage
  - BCP\_vec, [164](#)
- ensure\_explicit
  - BCP\_problem\_core\_change, [110](#)
- entry
  - BCP\_vec, [159](#)
- erase
  - BCP\_vec, [163](#)
- erase\_by\_index
  - BCP\_vec, [163](#)
- erase\_col\_set
  - BCP\_lp\_relax, [72](#)
- erase\_row\_set
  - BCP\_lp\_relax, [72](#)
- ExecutableName
  - BCP\_tm\_par, [132](#)
- fake\_objective\_value
  - BCP\_lp\_result, [74](#)
- fake\_objective\_values
  - BCP\_presolved\_lp\_brobj, [105](#)
- fathomable
  - BCP\_presolved\_lp\_brobj, [104](#)
- FeasibilityTest
  - BCP\_lp\_par, [62](#)
- finish
  - BCP\_vec, [164](#)
- FirstLP\_AllCutsTimeout
  - BCP\_lp\_par, [64](#)
- FirstLP\_AllVarsTimeout
  - BCP\_lp\_par, [64](#)
- FirstLP\_FirstCutTimeout
  - BCP\_lp\_par, [63](#)
- FirstLP\_FirstVarTimeout
  - BCP\_lp\_par, [64](#)
- forced\_cut\_bd
  - BCP\_lp\_branching\_object, [56](#)
- forced\_cut\_bd\_child
  - BCP\_lp\_branching\_object, [54](#)
- forced\_cut\_pos
  - BCP\_lp\_branching\_object, [55](#)
- forced\_var\_bd
  - BCP\_lp\_branching\_object, [56](#)
- forced\_var\_bd\_child
  - BCP\_lp\_branching\_object, [54](#)
- forced\_var\_pos
  - BCP\_lp\_branching\_object, [55](#)
- front
  - BCP\_vec, [160](#)
- generate\_cuts
  - BCP\_cg\_user, [34](#)
- generate\_cuts\_in\_lp
  - BCP\_lp\_user, [87](#)
- generate\_heuristic\_solution
  - BCP\_lp\_user, [84](#)
- generate\_vars
  - BCP\_vg\_user, [175](#)
- generate\_vars\_in\_lp
  - BCP\_lp\_user, [87](#)
- GenerateVars
  - BCP\_tm\_par, [130](#)
- get\_lower\_bounds
  - BCP\_presolved\_lp\_brobj, [104](#)

- get\_results
  - BCP\_lp\_result, [74](#)
  - BCP\_presolved\_lp\_brobj, [105](#)
- Granularity
  - BCP\_lp\_par, [63](#)
  - BCP\_tm\_par, [131](#)
- had\_numerical\_problems
  - BCP\_presolved\_lp\_brobj, [104](#)
- has\_ub
  - BCP\_cg\_prob, [29](#)
  - BCP\_vg\_prob, [170](#)
- implied\_cut\_bd\_child
  - BCP\_lp\_branching\_object, [54](#)
- implied\_var\_bd\_child
  - BCP\_lp\_branching\_object, [54](#)
- increase\_effective\_count
  - BCP\_cut, [41](#)
- index
  - BCP\_parameter, [98](#)
  - BCP\_vec, [159](#)
- IneffectiveBeforeDelete
  - BCP\_lp\_par, [62](#)
- IneffectiveConstraints
  - BCP\_lp\_par, [61](#)
- InfoForCG
  - BCP\_lp\_par, [63](#)
- InfoForVG
  - BCP\_lp\_par, [63](#)
- init\_new\_phase
  - BCP\_tm\_user, [139](#)
- init\_pos\_for\_added
  - BCP\_lp\_branching\_object, [55](#)
- initialize\_core
  - BCP\_tm\_user, [138](#)
- initialize\_int\_and\_sos\_list
  - BCP\_lp\_user, [82](#)
- initialize\_new\_search\_tree\_node
  - BCP\_lp\_user, [82](#)
- initialize\_solver\_interface
  - BCP\_lp\_user, [81](#)
- insert
  - BCP\_vec, [161](#)
- insert\_aux
  - BCP\_vec, [158](#)
- int\_params
  - BCP\_lp\_par, [61](#)
  - BCP\_tm\_par, [130](#)
  - BCP\_ts\_par, [140](#)
  - BCP\_vg\_par, [168](#)
- intAndSosObjects
  - BCP\_lp\_prob, [67](#)
- IntegerTolerance
  - BCP\_lp\_par, [63](#)
- is\_fixed
  - BCP\_var, [147](#)
- is\_fixed\_to\_zero
  - BCP\_var, [147](#)
- is\_non\_removable
  - BCP\_cut, [41](#)
  - BCP\_var, [147](#)
- is\_removable
  - BCP\_var, [147](#)
- is\_to\_be\_removed
  - BCP\_cut, [41](#)
  - BCP\_var, [147](#)
- keep
  - BCP\_vec, [162](#)
- keep\_by\_index
  - BCP\_vec, [162](#), [163](#)
- LPscheduler\_MaxNodeIdNum
  - BCP\_tm\_par, [131](#)
- LPscheduler\_MaxSbldNum
  - BCP\_tm\_par, [131](#)
- LPscheduler\_MinSbldNum
  - BCP\_tm\_par, [131](#)
- LPscheduler\_OverEstimationStatic
  - BCP\_tm\_par, [131](#)
- LaterLP\_AllCutsTimeout
  - BCP\_lp\_par, [64](#)
- LaterLP\_AllVarsTimeout
  - BCP\_lp\_par, [64](#)
- LaterLP\_FirstCutTimeout
  - BCP\_lp\_par, [63](#)
- LaterLP\_FirstVarTimeout
  - BCP\_lp\_par, [64](#)
- lb
  - BCP\_cut, [40](#)
  - BCP\_var, [146](#)
- lb\_multiplier
  - BCP\_tm\_prob, [134](#)
- load\_problem
  - BCP\_lp\_user, [82](#)
- LogFileName
  - BCP\_lp\_par, [64](#)
  - BCP\_tm\_par, [132](#)
- logical\_fixing
  - BCP\_lp\_user, [88](#)
- lower\_bound
  - BCP\_tm\_user, [137](#)
- LowerBound
  - BCP\_col, [36](#)
  - BCP\_row, [115](#)
- LpMachines
  - BCP\_tm\_par, [132](#)
- LpProcessNum
  - BCP\_tm\_par, [131](#)

LpVerb\_AddedCutCount  
     BCP\_lp\_par, 60  
 LpVerb\_AddedVarCount  
     BCP\_lp\_par, 60  
 LpVerb\_ChildrenInfo  
     BCP\_lp\_par, 60  
 LpVerb\_ColumnGenerationInfo  
     BCP\_lp\_par, 60  
 LpVerb\_CutsToCutPoolCount  
     BCP\_lp\_par, 60  
 LpVerb\_FathomInfo  
     BCP\_lp\_par, 60  
 LpVerb\_FinalRelaxedSolution  
     BCP\_lp\_par, 60  
 LpVerb\_First  
     BCP\_lp\_par, 60  
 LpVerb\_GeneratedCutCount  
     BCP\_lp\_par, 61  
 LpVerb\_GeneratedVarCount  
     BCP\_lp\_par, 61  
 LpVerb\_IterationCount  
     BCP\_lp\_par, 60  
 LpVerb\_Last  
     BCP\_lp\_par, 61  
 LpVerb\_LpMatrixSize  
     BCP\_lp\_par, 60  
 LpVerb\_LpSolutionValue  
     BCP\_lp\_par, 60  
 LpVerb\_MatrixCompression  
     BCP\_lp\_par, 60  
 LpVerb\_NodeTime  
     BCP\_lp\_par, 60  
 LpVerb\_PresolvePositions  
     BCP\_lp\_par, 60  
 LpVerb\_PresolveResult  
     BCP\_lp\_par, 60  
 LpVerb\_ProcessedNodeIndex  
     BCP\_lp\_par, 60  
 LpVerb\_RelaxedSolution  
     BCP\_lp\_par, 60  
 LpVerb\_ReportCutGenTimeout  
     BCP\_lp\_par, 60  
 LpVerb\_ReportLocalCutPoolSize  
     BCP\_lp\_par, 61  
 LpVerb\_ReportLocalVarPoolSize  
     BCP\_lp\_par, 61  
 LpVerb\_ReportVarGenTimeout  
     BCP\_lp\_par, 61  
 LpVerb\_RepricingResult  
     BCP\_lp\_par, 61  
 LpVerb\_RowEffectivenessCount  
     BCP\_lp\_par, 61  
 LpVerb\_StrongBranchPositions  
     BCP\_lp\_par, 61  
 LpVerb\_StrongBranchResult  
     BCP\_lp\_par, 61  
 LpVerb\_VarTightening  
     BCP\_lp\_par, 61  
 LpVerb\_VarsToVarPoolCount  
     BCP\_lp\_par, 60  
 lpres  
     BCP\_presolved\_lp\_brobj, 103  
 make\_active  
     BCP\_cut, 42  
     BCP\_var, 149  
 make\_fit  
     BCP\_buffer, 24  
 make\_non\_removable  
     BCP\_cut, 42  
     BCP\_var, 149  
 make\_to\_be\_removed  
     BCP\_cut, 42  
     BCP\_var, 149  
 make\_wrtcore\_if\_shorter  
     BCP\_problem\_core\_change, 111  
 matrix  
     BCP\_problem\_core, 107  
 MaxCutsAddedPerIteration  
     BCP\_lp\_par, 62  
 MaxHeapSize  
     BCP\_tm\_par, 131  
     BCP\_ts\_par, 140  
 MaxLeftoverCutFrac  
     BCP\_lp\_par, 63  
 MaxLeftoverCutNum  
     BCP\_lp\_par, 62  
 MaxNonDualFeasToAdd\_Frac  
     BCP\_lp\_par, 63  
 MaxNonDualFeasToAdd\_Max  
     BCP\_lp\_par, 62  
 MaxNonDualFeasToAdd\_Min  
     BCP\_lp\_par, 62  
 MaxPresolverIter  
     BCP\_lp\_par, 62  
 MaxRunTime  
     BCP\_lp\_par, 64  
     BCP\_tm\_par, 131  
 MaxVarsAddedPerIteration  
     BCP\_lp\_par, 62  
 MessagePassingIsSerial  
     BCP\_lp\_par, 60  
     BCP\_tm\_par, 130  
     BCP\_vg\_par, 168  
 modify\_lp\_parameters  
     BCP\_lp\_user, 82  
 move\_deletable\_to\_pool  
     BCP\_cut\_set, 47



- msg\_buf
  - BCP\_cg\_prob, 30
  - BCP\_vg\_prob, 171
- msg\_env
  - BCP\_cg\_prob, 30
  - BCP\_vg\_prob, 171
- msgenv\_init
  - USER\_initialize, 186
- msgtag
  - BCP\_buffer, 24
- multicast
  - BCP\_message\_environment, 94
  - BCP\_single\_environment, 120
- NiceLevel
  - BCP\_lp\_par, 61
  - BCP\_tm\_par, 131
  - BCP\_vg\_par, 168
- NoCompressionAtFathom
  - BCP\_lp\_par, 60
- node
  - BCP\_lp\_prob, 67
- node\_index
  - BCP\_cg\_prob, 31
  - BCP\_vg\_prob, 172
- node\_iteration
  - BCP\_cg\_prob, 31
  - BCP\_vg\_prob, 172
- node\_level
  - BCP\_cg\_prob, 31
  - BCP\_vg\_prob, 172
- num\_procs
  - BCP\_message\_environment, 96
- obj
  - BCP\_var, 146
- obj\_type
  - BCP\_cut, 40
  - BCP\_cut\_algo, 44
  - BCP\_cut\_core, 45
  - BCP\_var, 146
  - BCP\_var\_algo, 151
  - BCP\_var\_core, 152
- Objective
  - BCP\_col, 36
  - BCP\_lp\_relax, 71
- objective\_value
  - BCP\_solution, 122
  - BCP\_solution\_generic, 124
- operator=
  - BCP\_buffer, 24
  - BCP\_col, 37
  - BCP\_lp\_relax, 72
  - BCP\_problem\_core\_change, 110
  - BCP\_row, 115
  - BCP\_vec, 160
- operator[]
  - BCP\_vec, 160
- over\_ub
  - BCP\_lp\_user, 81
- pack
  - BCP\_buffer, 25, 26
  - BCP\_internal\_brobj, 51
  - BCP\_lp\_relax, 72
  - BCP\_parameter\_set, 101
  - BCP\_problem\_core, 107
  - BCP\_problem\_core\_change, 111
  - BCP\_vec\_change, 167
  - BCP\_warmstart\_basis, 180
  - BCP\_warmstart\_dual, 182
  - BCP\_warmstart\_primaldual, 185
- pack\_dual\_solution
  - BCP\_lp\_user, 85
- pack\_feasible\_solution
  - BCP\_lp\_user, 84
- pack\_module\_data
  - BCP\_tm\_user, 137
- pack\_primal\_solution
  - BCP\_lp\_user, 85
- pack\_size
  - BCP\_problem\_core\_change, 111
- packer
  - BCP\_cg\_prob, 30
  - BCP\_lp\_prob, 67
  - BCP\_tm\_prob, 134
  - BCP\_vg\_prob, 171
- par
  - BCP\_cg\_prob, 30
  - BCP\_vg\_prob, 171
- parent
  - BCP\_lp\_prob, 68
- parent\_process
  - BCP\_message\_environment, 94
  - BCP\_single\_environment, 119
- phase
  - BCP\_cg\_prob, 31
  - BCP\_vg\_prob, 172
- pi
  - BCP\_vg\_prob, 172
- pop\_back
  - BCP\_vec, 162
- primalTolerance
  - BCP\_lp\_result, 74
- print\_branching\_info
  - BCP\_lp\_branching\_object, 55
- probe
  - BCP\_message\_environment, 95
  - BCP\_single\_environment, 120

probe\_messages  
     BCP\_cg\_prob, 29  
     BCP\_vg\_prob, 171  
 process\_lp\_result  
     BCP\_lp\_user, 83  
 process\_message  
     BCP\_lp\_user, 81  
     BCP\_tm\_user, 138  
 purge\_slack\_pool  
     BCP\_lp\_user, 90  
 push\_back  
     BCP\_vec, 161  
  
 QualityRatioToAllowDiving\_HasUB  
     BCP\_tm\_par, 131  
 QualityRatioToAllowDiving\_NoUB  
     BCP\_tm\_par, 131  
  
 read\_from\_arglist  
     BCP\_parameter\_set, 101  
 read\_from\_file  
     BCP\_parameter\_set, 101  
 read\_from\_stream  
     BCP\_parameter\_set, 101  
 ReadRootCutsFrom  
     BCP\_tm\_par, 132  
 receive  
     BCP\_message\_environment, 95  
     BCP\_single\_environment, 120  
 receive\_cuts  
     BCP\_tm\_node\_to\_send, 128  
 receive\_node\_desc  
     BCP\_tm\_node\_to\_send, 128  
 receive\_vars  
     BCP\_tm\_node\_to\_send, 128  
 reduced\_cost\_fixing  
     BCP\_lp\_user, 89  
 register\_process  
     BCP\_message\_environment, 94  
     BCP\_single\_environment, 119  
 release\_sb\_id  
     BCP\_scheduler, 118  
 RemoveExploredBranches  
     BCP\_tm\_par, 130  
 replace\_solution  
     BCP\_tm\_user, 138  
 ReportWhenDefaultIsExecuted  
     BCP\_lp\_par, 60  
     BCP\_tm\_par, 130  
     BCP\_vg\_par, 168  
 request\_node\_id  
     BCP\_scheduler, 118  
 request\_sb\_ids  
     BCP\_scheduler, 117  
 reserve  
     BCP\_lp\_relax, 72  
     BCP\_vec, 160  
 restore\_feasibility  
     BCP\_lp\_user, 86  
 root\_node\_sent  
     BCP\_tm\_prob, 134  
 root\_pricing\_unpacked  
     BCP\_tm\_flags, 126  
 RowLowerBound  
     BCP\_lp\_relax, 71  
 RowUpperBound  
     BCP\_lp\_relax, 71  
 rownum  
     BCP\_lp\_relax, 71  
  
 SaveRootCutsTo  
     BCP\_tm\_par, 132  
 ScaleMatrix  
     BCP\_lp\_par, 61  
 select\_branching\_candidates  
     BCP\_lp\_user, 89  
 select\_fractions  
     BCP\_lp\_user, 81  
 select\_nonzeros  
     BCP\_lp\_user, 81  
 select\_positives  
     BCP\_lp\_user, 81  
 select\_zeros  
     BCP\_lp\_user, 81  
 send  
     BCP\_message\_environment, 94  
     BCP\_single\_environment, 120  
 send\_cut  
     BCP\_cg\_user, 33  
 send\_var  
     BCP\_vg\_user, 175  
 SendFathomedNodeDesc  
     BCP\_lp\_par, 60  
 sender  
     BCP\_buffer, 24  
     BCP\_cg\_prob, 31  
     BCP\_vg\_prob, 172  
 set\_actions\_for\_children  
     BCP\_lp\_user, 90  
 set\_bcpind  
     BCP\_cut, 42  
     BCP\_var, 148  
 set\_content  
     BCP\_buffer, 24  
 set\_effective\_count  
     BCP\_cut, 41  
 set\_lb  
     BCP\_cut, 41  
     BCP\_var, 148

- set\_lb\_ub
  - BCP\_cut\_set, [47](#)
  - BCP\_var, [148](#)
  - BCP\_var\_set, [153](#)
- set\_lb\_ub\_st
  - BCP\_cut\_set, [47](#)
  - BCP\_var\_set, [154](#)
- set\_lower\_bounds
  - BCP\_presolved\_lp\_brobj, [104](#)
- set\_obj
  - BCP\_var, [148](#)
- set\_objective\_value
  - BCP\_solution\_generic, [124](#)
- set\_objective\_values
  - BCP\_presolved\_lp\_brobj, [105](#)
- set\_position
  - BCP\_buffer, [24](#)
- set\_size
  - BCP\_buffer, [24](#)
- set\_status
  - BCP\_cut, [42](#)
  - BCP\_var, [148](#)
- set\_ub
  - BCP\_cut, [41](#)
  - BCP\_var, [148](#)
- set\_user\_data\_for\_children
  - BCP\_lp\_user, [90](#)
- set\_var\_type
  - BCP\_var, [148](#)
- setParams
  - BCP\_scheduler, [117](#)
- size
  - BCP\_buffer, [24](#)
  - BCP\_vec, [159](#)
- SlackCutDiscardingStrategy
  - BCP\_lp\_par, [61](#)
- start
  - BCP\_vec, [164](#)
- start\_process
  - BCP\_message\_environment, [95](#)
  - BCP\_single\_environment, [120](#), [121](#)
- start\_processes
  - BCP\_message\_environment, [95](#), [96](#)
  - BCP\_single\_environment, [121](#)
- status
  - BCP\_cut, [40](#)
  - BCP\_var, [147](#)
- storage
  - BCP\_problem\_core\_change, [110](#)
  - BCP\_warmstart, [176](#)
  - BCP\_warmstart\_basis, [178](#)
  - BCP\_warmstart\_dual, [181](#)
  - BCP\_warmstart\_primaldual, [184](#)
- storage\_size
  - BCP\_vec\_change, [167](#)
  - BCP\_warmstart, [177](#)
  - BCP\_warmstart\_basis, [179](#)
  - BCP\_warmstart\_dual, [182](#)
  - BCP\_warmstart\_primaldual, [184](#)
- str\_array\_params
  - BCP\_lp\_par, [64](#)
  - BCP\_tm\_par, [132](#)
- str\_params
  - BCP\_lp\_par, [64](#)
  - BCP\_tm\_par, [131](#)
- StrongBranch\_CloseToHalfNum
  - BCP\_lp\_par, [62](#)
- StrongBranchNum
  - BCP\_lp\_par, [62](#)
- swap
  - BCP\_problem\_core\_change, [111](#)
  - BCP\_vec, [160](#)
- TerminationGap\_Absolute
  - BCP\_tm\_par, [131](#)
- TerminationGap\_Relative
  - BCP\_tm\_par, [131](#)
- test\_binary
  - BCP\_lp\_user, [84](#)
- test\_feasibility
  - BCP\_lp\_user, [83](#)
- test\_full
  - BCP\_lp\_user, [84](#)
- test\_integral
  - BCP\_lp\_user, [84](#)
- TmTimeout
  - BCP\_tm\_par, [131](#)
- TmVerb\_AllFeasibleSolution
  - BCP\_tm\_par, [130](#)
- TmVerb\_AllFeasibleSolutionValue
  - BCP\_tm\_par, [130](#)
- TmVerb\_BestFeasibleSolution
  - BCP\_tm\_par, [130](#)
- TmVerb\_BetterFeasibleSolution
  - BCP\_tm\_par, [130](#)
- TmVerb\_BetterFeasibleSolutionValue
  - BCP\_tm\_par, [130](#)
- TmVerb\_FinalStatistics
  - BCP\_tm\_par, [130](#)
- TmVerb\_First
  - BCP\_tm\_par, [130](#)
- TmVerb\_Last
  - BCP\_tm\_par, [130](#)
- TmVerb\_NewPhaseStart
  - BCP\_tm\_par, [130](#)
- TmVerb\_PrunedNodeInfo
  - BCP\_tm\_par, [130](#)
- TmVerb\_ReportDefault

- BCP\_tm\_par, [130](#)
- TmVerb\_SingleLineInfoFrequency
  - BCP\_tm\_par, [131](#)
- TmVerb\_TimeOfImprovingSolution
  - BCP\_tm\_par, [130](#)
- TmVerb\_TrimmedNum
  - BCP\_tm\_par, [130](#)
- TreeSearchStrategy
  - BCP\_tm\_par, [131](#)
- TrimTreeBeforeNewPhase
  - BCP\_tm\_par, [130](#)
- type
  - BCP\_parameter, [98](#)
- USER\_initialize, [185](#)
  - msgenv\_init, [186](#)
- ub
  - BCP\_cg\_prob, [29](#)
  - BCP\_cut, [40](#)
  - BCP\_var, [146](#)
  - BCP\_vg\_prob, [170](#)
- unchecked\_erase\_by\_index
  - BCP\_vec, [163](#)
- unchecked\_keep\_by\_index
  - BCP\_vec, [163](#)
- unchecked\_push\_back
  - BCP\_vec, [161](#)
- unchecked\_update
  - BCP\_vec, [162](#)
- UnconditionalDiveProbability
  - BCP\_tm\_par, [131](#)
- unpack
  - BCP\_buffer, [25, 26](#)
  - BCP\_internal\_brobj, [51](#)
  - BCP\_lp\_relax, [72](#)
  - BCP\_parameter\_set, [101](#)
  - BCP\_problem\_core, [107](#)
  - BCP\_problem\_core\_change, [111](#)
  - BCP\_vec\_change, [168](#)
- unpack\_cut
  - BCP\_vg\_prob, [171](#)
- unpack\_dual\_solution
  - BCP\_vg\_user, [175](#)
- unpack\_feasible\_solution
  - BCP\_tm\_user, [137](#)
- unpack\_module\_data
  - BCP\_cg\_user, [33](#)
  - BCP\_lp\_user, [81](#)
  - BCP\_ts\_user, [142](#)
  - BCP\_vg\_user, [175](#)
- unpack\_primal\_solution
  - BCP\_cg\_user, [33](#)
- unpack\_var
  - BCP\_cg\_prob, [29](#)
- update
  - BCP\_problem\_core\_change, [111](#)
  - BCP\_vec, [162](#)
  - BCP\_vec\_change, [167](#)
  - BCP\_warmstart, [176](#)
  - BCP\_warmstart\_basis, [179](#)
  - BCP\_warmstart\_dual, [182](#)
  - BCP\_warmstart\_primaldual, [184](#)
- upper\_bound
  - BCP\_cg\_prob, [30](#)
  - BCP\_vg\_prob, [172](#)
- UpperBound
  - BCP\_col, [36, 37](#)
  - BCP\_row, [115](#)
  - BCP\_tm\_par, [131](#)
- user
  - BCP\_cg\_prob, [29](#)
  - BCP\_vg\_prob, [171](#)
- user\_class
  - BCP\_user\_pack, [143](#)
- user\_data
  - BCP\_lp\_node, [58](#)
  - BCP\_presolved\_lp\_brobj, [104](#)
- var\_bounds\_child
  - BCP\_internal\_brobj, [50](#)
- var\_ch
  - BCP\_problem\_core\_change, [112](#)
- var\_pos
  - BCP\_problem\_core\_change, [112](#)
- var\_positions
  - BCP\_internal\_brobj, [50](#)
- var\_type
  - BCP\_var, [146](#)
- VarPoolCheckFrequency
  - BCP\_lp\_par, [61](#)
- varnum
  - BCP\_problem\_core, [107](#)
  - BCP\_problem\_core\_change, [110](#)
- vars
  - BCP\_cg\_prob, [30](#)
  - BCP\_problem\_core, [107](#)
- vars\_added
  - BCP\_lp\_branching\_object, [54](#)
- vars\_affected
  - BCP\_lp\_branching\_object, [53](#)
- vars\_to\_add
  - BCP\_lp\_branching\_object, [55](#)
- vars\_to\_cols
  - BCP\_lp\_user, [86](#)
- VerbosityShutUp
  - BCP\_tm\_par, [130](#)
- VgMachines
  - BCP\_tm\_par, [132](#)

VgProcessNum  
    BCP\_tm\_par, [131](#)  
VpMachines  
    BCP\_tm\_par, [132](#)  
VpProcessNum  
    BCP\_tm\_par, [131](#)  
  
WarmstartInfo  
    BCP\_lp\_par, [62](#)  
    BCP\_tm\_par, [131](#)  
warmstartRoot  
    BCP\_lp\_prob, [68](#)  
write\_to\_stream  
    BCP\_parameter\_set, [101](#)  
  
x  
    BCP\_cg\_prob, [30](#)