Refreshing C for SCIP Programming

Benjamin Müller

Zuse Institute Berlin

28th of September, 2015

Research Center MATHEON
Mathematics for Key Technologies

Berlin
Mathematical
School

MODAL
Mathematical Optimization and Data Analysis Laboratories

Refreshing C for SCIP Programming

C Basics

Pointers

Structs, Enums, and Macros

Arrays and SCIP's Memory Management

First IP with SCIP

Refreshing C for SCIP Programming

Hello World

```c
#include <stdio.h>          /* include system library */

/* This is a C comment.
 * Every C program has exactly one "main" function.
 */
int main()
{
  printf("hello world\n"); /* prints 'hello world' to the shell */

  return 0;
}
```

Hello World

```c
#include <stdio.h>          /* include system library */

/* This is a C comment.
 * Every C program has exactly one "main" function.
 */
int main()
{
  printf("hello world\n"); /* prints 'hello world' to the shell */

  return 0;
}
```

- compile with: `gcc <name>.c`
- run with `./a.out`
- other C/C++ compilers: MSVC, gcc, intel, clang

Types

integer types

- `char`
- `short`
- `int`
- `long`
- `long long`

use `char` to store single characters, e.g., `'A'`, `'3'`, `'$'`, `...`

floating point types

- `float`
- `double` (2× the precision of a `float`)

Note: no `boolean` type, no `string` type

Operators

Arithmetic Operators

`+ - * /`	arithmetic operators	`a + b`
`+= -= *= /=`	'adds to', ...	`a += b`
`++`	increment by one	`++a`

Comparison Operators

`==`	'equal'	`a == b`
`!=`	'not equal'	`a != b`
`>=`	'greater or equal'	`a >= b`
`<=`	'less or equal'	`a <= b`
`> <`	'strict greater/less'	`a < b` or `a > b`

Logic Operators

`!`	logic not	`!(a >= b)`
`&&`	logic and	`(a < b) && (b > 0)`
`||`	logic or	`(a > b) || (b < 0)`

if Statement

```c
int a;                  /* declare variables */
int b = 2;              /* directly initialize */

a = 3;                  /* assign new value */

if( a >= b )
{
  /* enter block if condition is true */
}
else if( b == 2 )
{
  /* enter block if second condition is true and the first is not */
}
else
{
  /* enter block if all conditions above were false */
}
```

Loops

for - loop

```
1  int i;
2
3  /*
4   * for( 'start value'; 'continue condition'; 'increase value' )
5   */
6  for( i = 0; i < 10; ++i ) /* start with i = 0 */
7  {
8    /* enter block if condition is (still) true */
9    ...;
10   /* jump back to line 4 and increment i by one */
11 }
```

while - loop

```
1  int i = 0;
2
3  /*
4   * while( 'continue condition' )
5   */
6  while( i < 10 )
7  {
8    /* enter block if condition is (still) true */
9
10   ++i; /* infinite loop if i does not increase */
11 }
```

Format Specifiers

How to use format specifiers:

- C documentation: `int printf ( const char * format, ... );`
- `\n` for a line break
- `\t` for a tab

```c
#include <stdio.h>          /* include system library */

int main()
{
  char c = 'c';             /* specifier: %c */
  short s = 1;              /* specifier: %i */
  int i = 2;                /* specifier: %d */
  long l = 3;               /* specifier: %ld */
  long long ll = 4;         /* specifier: %lld */
  float f = 5.0;            /* specifier: %e or %f */
  double d = 1.0 / 3.0;     /* specifier: %e or %f */

  printf("%i + %d = %d\n", s, i, (s + i)); /* prints 1 + 2 = 3 */

  printf("%1.4e\n", d);     /* prints 3.3333e-01 */
  printf("%1.4f\n", d);     /* prints 0.3333 */

  return 0;
}
```

Functions

A function is a segment that groups code to perform a specific task.

```c
/* return-type function-name ( arg_type arg1, ..., arg_type argN ); */
double foo(int a, double b)          /* function header */
{
  double c;          /* c only exists in this function (scope) */

  c = a * b;         /* assign value to c */

  return c;          /* has to end with a return ...; */
}
```

Functions

A function is a segment that groups code to perform a specific task.

```c
/* return-type function-name ( arg_type arg1, ..., arg_type argN ); */
double foo(int a, double b)          /* function header */
{
  double c;          /* c only exists in this function (scope) */

  c = a * b;         /* assign value to c */

  return c;          /* has to end with a return ...; */
}
```

```c
/* function does not have a return value nor parameters */
void foofoo()
{
  double c;          /* this c has nothing to do with the c in foo(...) */

  c = foo(3, 2.5);   /* call functions foo(...) */

  if( c == 0.0 )
    return;          /* terminates the function immediately */

  printf("foo(3,2.5) is %e\n", c); /* prints 'foo(3,2.5) = 7.5 */
}
```

Header and Source Files

c_one.c

```c
#include <stdio.h>                /* <name> for system libraries */
#include <assert.h>

#include "c_one.h"                /* "name" for our header */

/* returns the sum of integers between a and b */
double sumup(int a, int b)
{
  int i;                          /* declare variables */
  double sum = 0.0;               /* variables can be initialized */

  assert(a >= 0 && b >= 0);       /* assert to state preconditions */

  if( b < a )                     /* execute code */
  {
    int t;                        /* exists between the curly braces (scope) */
    t = a; a = b; b = t;          /* exchange values of a and b */
  }
  assert(a <= b);

  for( i = a; i <= b; ++i )       /* loop from a to b */
    sum += i;                     /* add i to sum */

  return sum;                     /* return value of the function */
}
```

Header and Source Files

c_one.h

```c
/* declare interface function here but implement it in another *.c file */
extern double sumup(int a, int b);
```

c_main.c

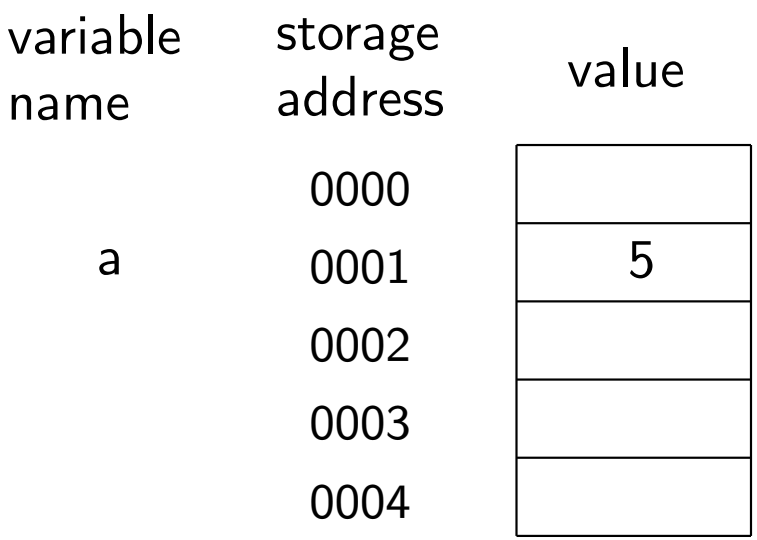
```c
#include <stdio.h>              /* <name> for system library */
#include "c_one.h"              /* "name" for our header */

int main()
{
  printf("sumup(2,6) = %e\n", sumup(2,6)); /* prints 'sumup(2,6) = 20' */
  return 0;
}
```

- header files contain function declarations and macro definitions
- `#include "name".h` copies the whole content of the header file
- use them to structure your project and define interfaces

Refreshing C for SCIP Programming

Pointers

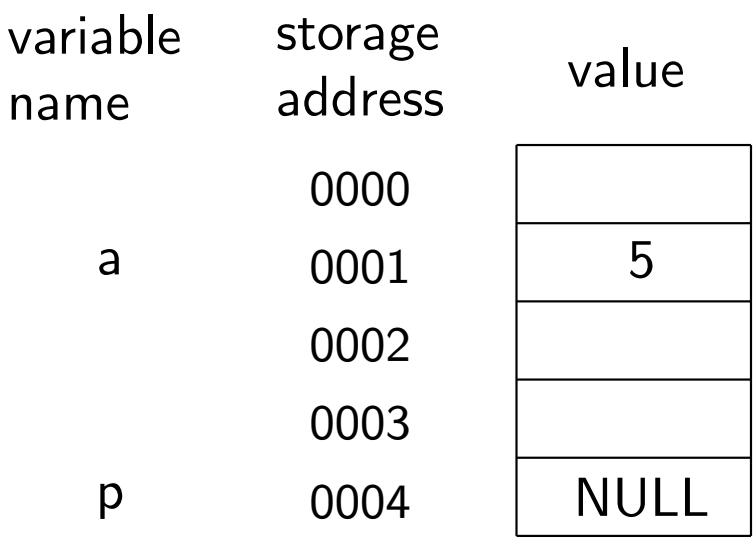
A pointer is an object, whose value refers to another value stored elsewhere using its address.




```c
int a = 5;
```

Pointers

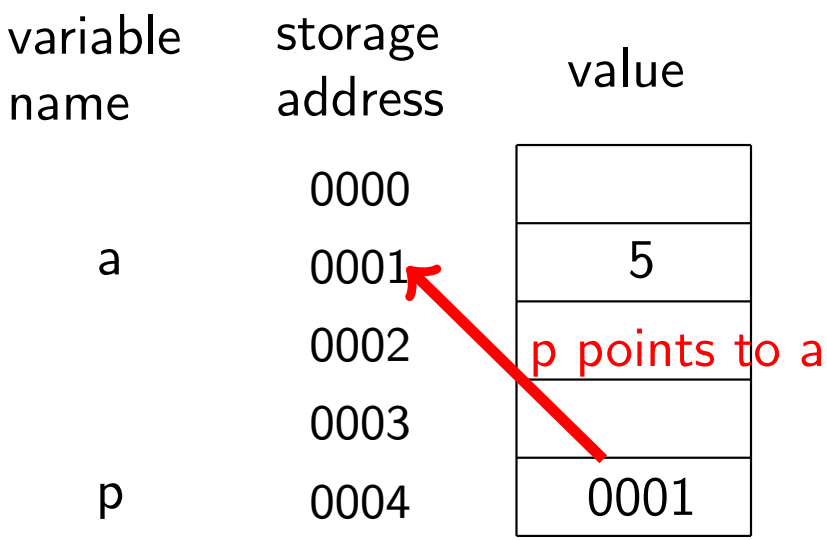
A pointer is an object, whose value refers to another value stored elsewhere using its address.




```
1  int a = 5;
2  int* p = NULL;          /* p is a pointer, p points nowhere */
```

Pointers

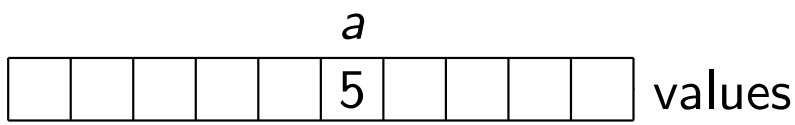
A pointer is an object, whose value refers to another value stored elsewhere using its address.




```c
int a = 5;
int* p = NULL;          /* p is a pointer, p points nowhere */

p = &a;                 /* store address of a into p */
```

Pointers

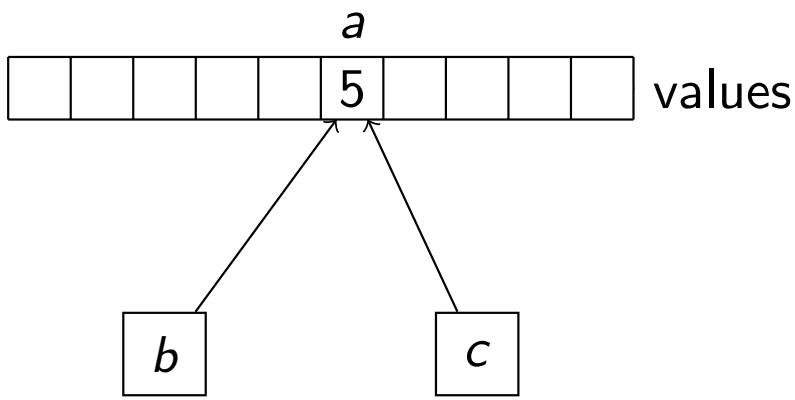
Memory




```
1  int a = 5;
```

Pointers

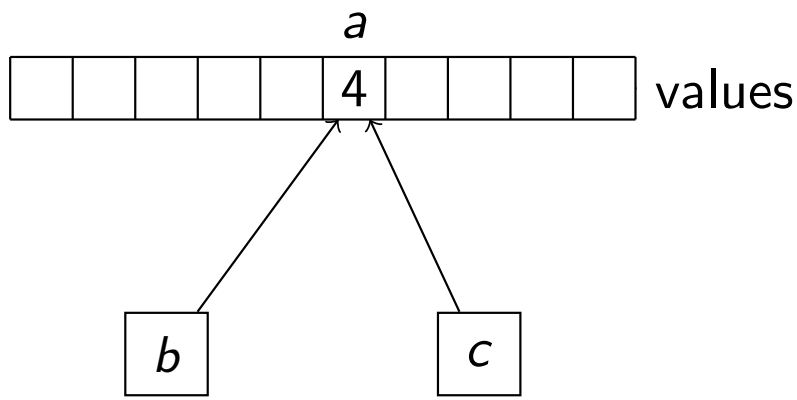
Memory




```c
int a = 5;
int* b = &a;          /* store address of var in pointer variable */
int* c = b;           /* create a copy of the pointer */

printf("%u\n", (&a == c));          /* prints 1 */
```

Pointers

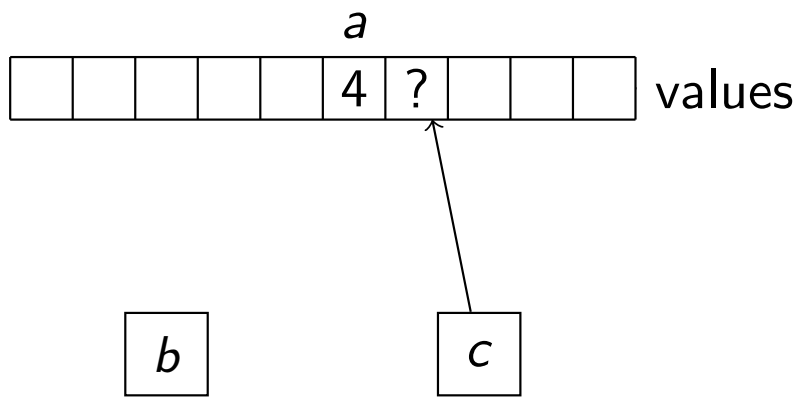
Memory




```c
int a = 5;
int* b = &a;          /* store address of var in pointer variable */
int* c = b;           /* create a copy of the pointer */

printf("%u\n", (&a == c));          /* prints 1 */

*c = 4;               /* manipulate value through pointer */
printf("%d, %d, %d\n", a, *b, *c); /* prints 4, 4, 4 */
```

Pointers

Memory




```c
int a = 5;
int* b = &a;          /* store address of var in pointer variable */
int* c = b;           /* create a copy of the pointer */

printf("%u\n", (&a == c));          /* prints 1 */

*c = 4;               /* manipulate value through pointer */
printf("%d, %d, %d\n", a, *b, *c); /* prints 4, 4, 4 */

b = NULL;                           /* b points nowhere */
printf("%d\n", *b);                 /* segmentation fault */

c = c + 1;                          /* manipulate address of pointer */
printf("%d\n", *c);                 /* undefined behavior */
```

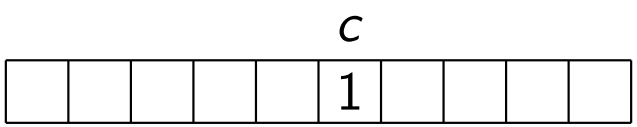
Pointers

```c
void increment(int b)
{
  b += 1;
}

int main()
{
  int c = 1;

  increment(c);
  printf("%d\n", c);      /* prints 1 */

  return 0;
}
```

Pointers

```c
void increment(int b)
{
  b += 1;
}

int main()
{
  int c = 1;

  increment(c);
  printf("%d\n", c);      /* prints 1 */

  return 0;
}
```

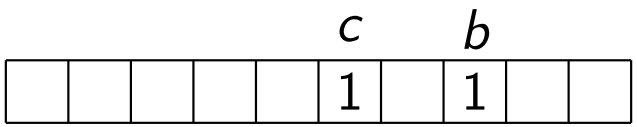
Why? b is just a copy of c (call-by-value).



Pointers

```c
void increment(int b)
{
  b += 1;
}

int main()
{
  int c = 1;

  increment(c);
  printf("%d\n", c);      /* prints 1 */

  return 0;
}
```

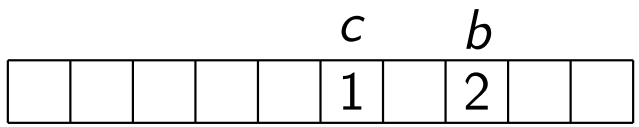
Why? b is just a copy of c (call-by-value).



Pointers

```c
void increment(int b)
{
  b += 1;
}

int main()
{
  int c = 1;

  increment(c);
  printf("%d\n", c);      /* prints 1 */

  return 0;
}
```

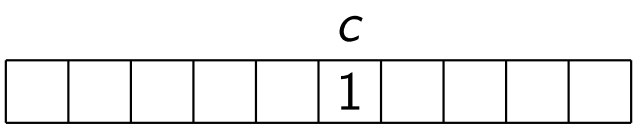
Why? b is just a copy of c (call-by-value).



Pointers

```c
void increment(int b)
{
  b += 1;
}

int main()
{
  int c = 1;

  increment(c);
  printf("%d\n", c);      /* prints 1 */

  return 0;
}
```

Why? b is just a copy of c (call-by-value).

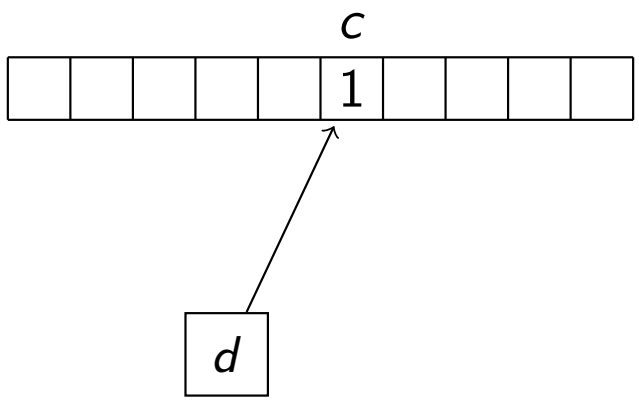


Pointers

```c
void increment(int* b)
{
  *b += 1;                /* segmentation fault if b is NULL */
}

int main()
{
  int c = 1;
  int* d = &c;

  increment(d);           /* also possible to use &c here */
  printf("%d\n", c);      /* prints 2 */

  return 0;
}
```

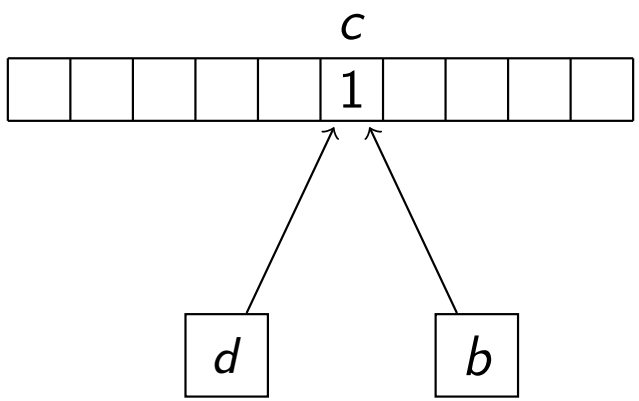
Pointers

```c
void increment(int* b)
{
  *b += 1;                  /* segmentation fault if b is NULL */
}

int main()
{
  int c = 1;
  int* d = &c;

  increment(d);             /* also possible to use &c here */
  printf("%d\n", c);        /* prints 2 */

  return 0;
}
```



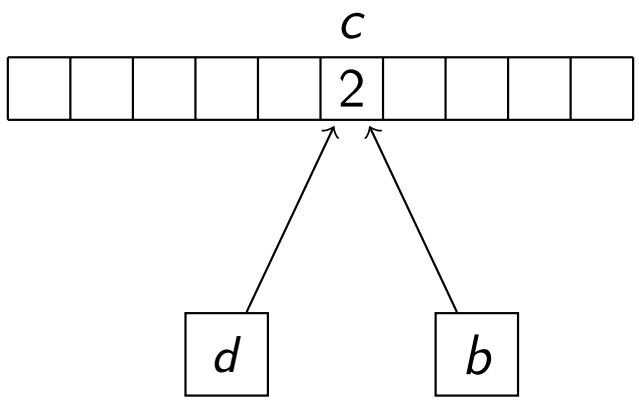
Pointers

```c
void increment(int* b)
{
  *b += 1;                  /* segmentation fault if b is NULL */
}

int main()
{
  int c = 1;
  int* d = &c;

  increment(d);             /* also possible to use &c here */
  printf("%d\n", c);        /* prints 2 */

  return 0;
}
```



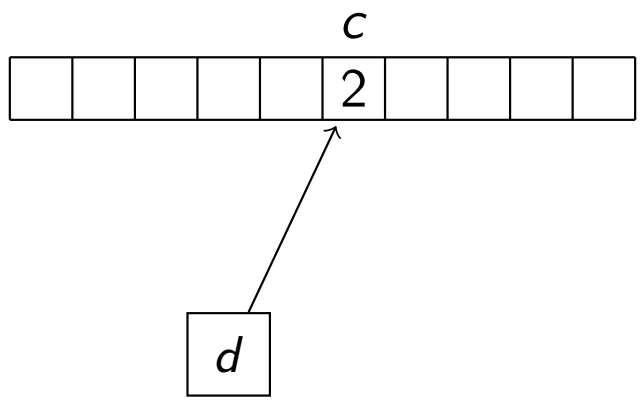
Pointers

```c
void increment(int* b)
{
  *b += 1;                  /* segmentation fault if b is NULL */
}

int main()
{
  int c = 1;
  int* d = &c;

  increment(d);             /* also possible to use &c here */
  printf("%d\n", c);        /* prints 2 */

  return 0;
}
```



Pointers

```c
void increment(int* b)
{
  *b += 1;                  /* segmentation fault if b is NULL */
}

int main()
{
  int c = 1;
  int* d = &c;

  increment(d);             /* also possible to use &c here */
  printf("%d\n", c);        /* prints 2 */

  return 0;
}
```



Pointers

```
1  /* use pointers for returning more than just one single value */
2  double calc(double x, int* success)
3  {
4    if( x != 0.0 )
5    {
6      *success = 1;          /* segmentation fault if success is NULL */
7      return 1.0 / x;
8    }
9
10   *success = 0;            /* store 0 if something went wrong */
11
12   return 0.0;
13 }
```

Pointers

```c
/* use pointer to return more values */
double calc(double x, int* success)
{
  if( x != 0.0 )
  {
    *success = 1;          /* segmentation fault if success is NULL */
    return 1.0 / ;
  }

  *success = 0;            /* store 0 if something went wrong */

  return 0.0;
}

int main()
{
  double result;
  int success;

  result = calc(2.0, &success);    /* note: success is not initialized */

  if( success == 1 )
    printf("calc(2.0) = %f\n", result);   /* prints 'calc(2) = 0.5' */
  else
    printf("division by zero is not allowed\n");

  return 0;
}
```

Refreshing C for SCIP Programming

C Basics

Pointers

Structs, Enums, and Macros

Arrays and SCIP's Memory Management

First IP with SCIP

Structs

```c
/* declaration of a listed group of variables */
struct Data
{
  double a;
  int b;
};
typedef struct Data DATA;          /* adds a shortcut name */
```

Structs

```c
/* declaration of a listed group of variables */
struct Data
{
  double a;
  int b;
};
typedef struct Data DATA;          /* adds a shortcut name */

int main()
{
  DATA d;                          /* memory allocated in its scope */
  DATA* p;                         /* pointer to a struct */

  /* access to the variables of a struct */
  d.a = 0.0;
  d.b = -1;

  /* pointer with structs */
  p = &d;                          /* p points to d */
  (*p).a = 2.0;                    /* change value through the pointer */
  p->a = 3.0;                      /* the same as (*p).a = 3.0 */
  printf("%d %e\n", d.a, d.b);     /* prints 2.0 -1 */

  return 0;
}
```

Structs

```
1  /* call-by-value: the whole struct gets copied here */
2  void foo1(DATA d)
3  {
4    d.a = 1;
5  }
6
7  /* call-by-value: pointer gets copied here */
8  void foo2(DATA* d)
9  {
10   (*d).a = 1;              /* the same as d->a = 1; */
11 }
```

Structs

```c
/* call-by-value: the whole struct gets copied here */
void foo1(DATA d)
{
  d.a = 1;
}

/* call-by-value: pointer gets copied here */
void foo2(DATA* d)
{
  (*d).a = 1;              /* the same as d->a = 1; */
}

int main()
{
  DATA d;

  d.a = 0;

  foo1(d);
  printf("%d\n", d.a);    /* prints 0 */

  foo2(&d);
  printf("%d\n", d.a);    /* prints 1 */

  return 0;
}
```

Structs

```
1  /** SCIP main data structure */
2  struct Scip
3  {
4     /* INIT */
5     SCIP_MEM*           mem;           /* block memory buffers */
6     SCIP_SET*           set;           /* global SCIP settings */
7     SCIP_INTERRUPT*     interrupt;     /* CTRL-C interrupt data */
8     SCIP_DIALOGHDLR*    dialoghdlr;    /* dialog handler for user interface */
9     SCIP_MESSAGEHDLR*   messagehdlr;   /* message handler for output handling */
10    SCIP_CLOCK*         totaltime;     /* total SCIP running time */
11
12    /* PROBLEM */
13    SCIP_STAT*          stat;          /* dynamic problem statistics */
14    SCIP_PROB*          origprob;      /* original problem data */
15    SCIP_PRIMAL*        origprimal;    /* primal data and solution storage */
16
17    ...;
18 };
19
20 typedef struct Scip SCIP;
```

- structs can contain other structs (also pointer to other structs)
- most functions in SCIP need a SCIP* pointer

Enums

An enum is a data type consisting of a set of named values.

```
1  /** return codes for SCIP functions: non-positive return codes are errors */
2  enum SCIP_Retcode
3  {
4     SCIP_OKAY                =  +1,  /* normal termination */
5     SCIP_ERROR               =   0,  /* unspecified error */
6     SCIP_NOMEMORY            =  -1,  /* insufficient memory error */
7     SCIP_READERROR           =  -2,  /* read error */
8     SCIP_WRITEERROR          =  -3,  /* write error */
9     ...;
10 };
11 typedef enum SCIP_Retcode SCIP_RETCODE;
```

SCIP_RETCODEs ...

- are returned by most of the SCIP functions
- should be handled with SCIP_CALL(...) macro

Macros

Macros are named code fragments for easier reuse in the project.

```
1  #define SCIP_Real double          /* type used for floating point values */
2  #define SCIP_BOOL unsigned int    /* type used for boolean values */
3  #define TRUE  1                   /* boolean value TRUE */
4  #define FALSE 0                   /* boolean value FALSE */
```

Macros

Macros are named code fragments for easier reuse in the project.

```
#define SCIP_Real double            /* type used for floating point values */
#define SCIP_BOOL unsigned int      /* type used for boolean values */
#define TRUE  1                     /* boolean value TRUE */
#define FALSE 0                     /* boolean value FALSE */
```

can have (several) arguments:

```
/* returns the square of a number, example: a = SQR(b); */
#define SQR(x)        ((x)*(x))

/* returns the retcode if it is not SCIP_OKAY */
#define SCIP_CALL(x)   do
                       {
                          SCIP_RETCODE _restat_;
                          if( (_restat_ = (x)) != SCIP_OKAY )
                          {
                             SCIPerrorMessage("Error <%d>\n", _restat_);
                             return _restat_;
                          }
                       }
                       while( FALSE )
```

Macros

Macros are named code fragments for easier reuse in the project.

```
1  #define SCIP_Real double            /* type used for floating point values */
2  #define SCIP_BOOL unsigned int      /* type used for boolean values */
3  #define TRUE  1                     /* boolean value TRUE */
4  #define FALSE 0                     /* boolean value FALSE */
```

can have (several) arguments:

```
1  /* returns the square of a number, example: a = SQR(b); */
2  #define SQR(x)        ((x)*(x))
3
4  /* returns the retcode if it is not SCIP_OKAY */
5  #define SCIP_CALL(x)   do
6                         {
7                            SCIP_RETCODE _restat_;
8                            if( (_restat_ = (x)) != SCIP_OKAY )
9                            {
10                              SCIPerrorMessage("Error <%d>\n", _restat_);
11                              return _restat_;
12                           }
13                        }
14                        while( FALSE )
```

- macros will be replaced by the preprocessor
- be cautious when using/defining macros

SCIP Retcodes

How to call SCIP functions.

```
1  /* transforms, presolves, and solves the problem */
2  SCIP_RETCODE SCIPsolve(SCIP* scip);
```

```
1  void foo(SCIP* scip)
2  {
3    SCIPsolve(scip);
4  }
```

SCIP Retcodes

How to call SCIP functions.

```
1  /* transforms, presolves, and solves the problem */
2  SCIP_RETCODE SCIPsolve(SCIP* scip);
```

```
1  void foo(SCIP* scip)
2  {
3    SCIPsolve(scip);
4  }
```

WRONG!

- does not use return value of `SCIPsolve`, we would not recognize any error

---

SCIP Retcodes

How to call SCIP functions.

```
1  /* transforms, presolves, and solves the problem */
2  SCIP_RETCODE SCIPsolve(SCIP* scip);
```

```
1  void foo(SCIP* scip)
2  {
3    SCIPsolve(scip);
4  }
```

WRONG!

- does not use return value of `SCIPsolve`, we would not recognize any error

---

```
1  void foo(SCIP* scip)
2  {
3    SCIP_CALL( SCIPsolve(scip) );
4  }
```

SCIP Retcodes

How to call SCIP functions.

```
1  /* transforms, presolves, and solves the problem */
2  SCIP_RETCODE SCIPsolve(SCIP* scip);
```

```
1  void foo(SCIP* scip)
2  {
3    SCIPsolve(scip);
4  }
```

WRONG!

- does not use return value of `SCIPsolve`, we would not recognize any error

---

```
1  void foo(SCIP* scip)
2  {
3    SCIP_CALL( SCIPsolve(scip) );
4  }
```

WRONG!

- in `SCIP_CALL(...)` we might return a `SCIP_RETCODE` but function is `void`

SCIP Retcodes

How to call SCIP functions.

```
1  /* transforms, presolves, and solves the problem */
2  SCIP_RETCODE SCIPsolve(SCIP* scip);
```

```
1  void foo(SCIP* scip)
2  {
3    SCIPsolve(scip);
4  }
```

WRONG!

- does not use return value of `SCIPsolve`, we would not recognize any error

---

```
1  void foo(SCIP* scip)
2  {
3    SCIP_CALL( SCIPsolve(scip) );
4  }
```

WRONG!

- in `SCIP_CALL(...)` we might return a `SCIP_RETCODE` but function is `void`

```
1  SCIP_RETCODE foo(SCIP* scip)
2  {
3    SCIP_CALL( SCIPsolve(scip) );
4    return SCIP_OKAY;
5  }
```

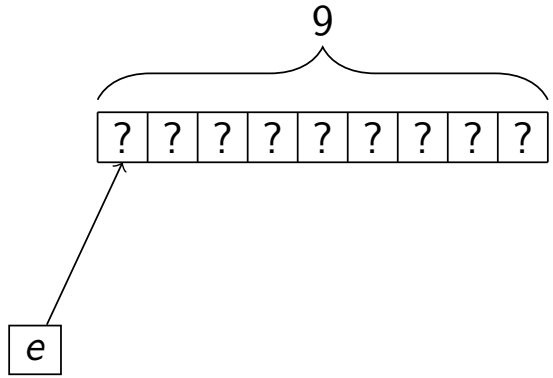
CORRECT!

- function needs to return a `SCIP_RETCODE` in any case

Refreshing C for SCIP Programming

Arrays

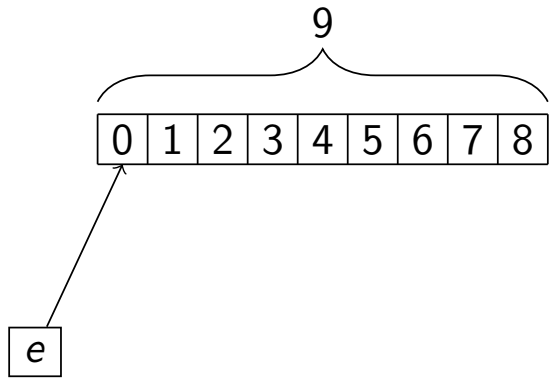
An array is a data structure consisting of elements of the same type.




```
1  int e[9];              /* defines an array of size 9; memory is not cleared! */
2  int* b;
3  int i;
```

Arrays

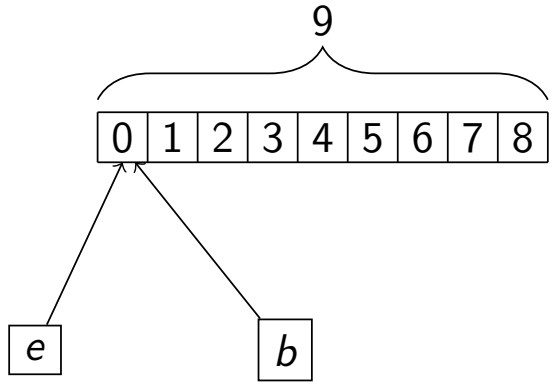
An array is a data structure consisting of elements of the same type.




```
1  int e[9];                /* defines an array of size 9; memory is not cleared! */
2  int* b;
3  int i;
4
5  for( i = 0; i < 9; ++i )
6    e[i] = i;              /* access value of i-th entry */
```

Arrays

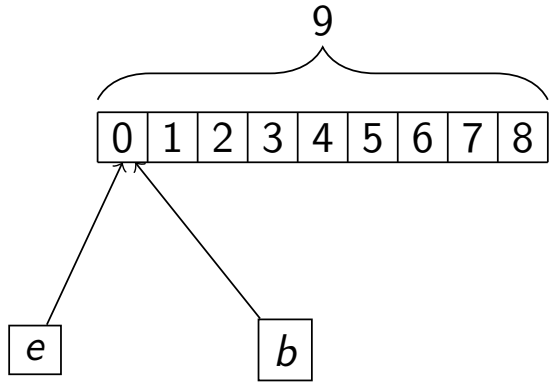
An array is a data structure consisting of elements of the same type.




```
1  int e[9];                /* defines an array of size 9; memory is not cleared! */
2  int* b;
3  int i;
4
5  for( i = 0; i < 9; ++i )
6    e[i] = i;              /* access value of i-th entry */
7
8  b = e;                   /* b points to the array */
9  b = &e[0];               /* nothing has changed */
```

Arrays

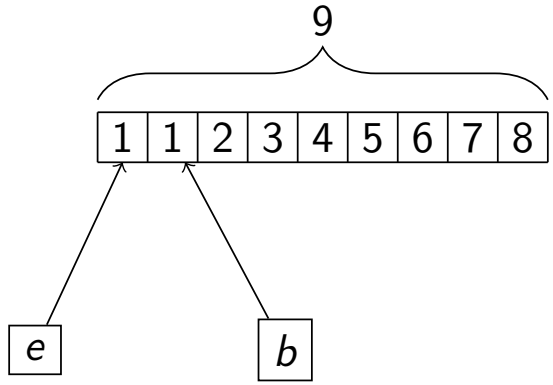
An array is a data structure consisting of elements of the same type.




```
1  int e[9];                /* defines an array of size 9; memory is not cleared! */
2  int* b;
3  int i;
4
5  for( i = 0; i < 9; ++i )
6    e[i] = i;              /* access value of i-th entry */
7
8  b = e;                   /* b points to the array */
9  b = &e[0];               /* nothing has changed */
10
11 for( i = 0; i < 4; ++i )
12 {
13   (*b) += 1;             /* increment value */
14   b += 1;                /* move pointer */
15 }
```

Arrays

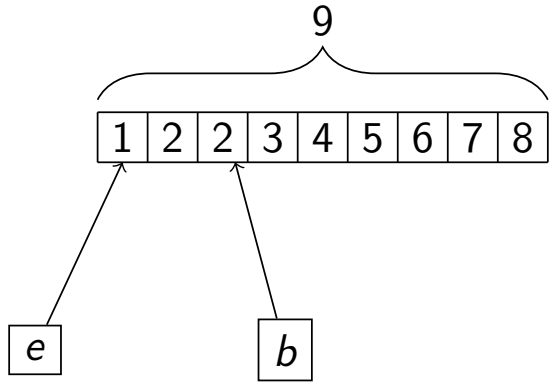
An array is a data structure consisting of elements of the same type.




```
1  int e[9];                /* defines an array of size 9; memory is not cleared! */
2  int* b;
3  int i;
4
5  for( i = 0; i < 9; ++i )
6    e[i] = i;              /* access value of i-th entry */
7
8  b = e;                   /* b points to the array */
9  b = &e[0];               /* nothing has changed */
10
11 for( i = 0; i < 4; ++i )
12 {
13   (*b) += 1;             /* increment value */
14   b += 1;                /* move pointer */
15 }
```

Arrays

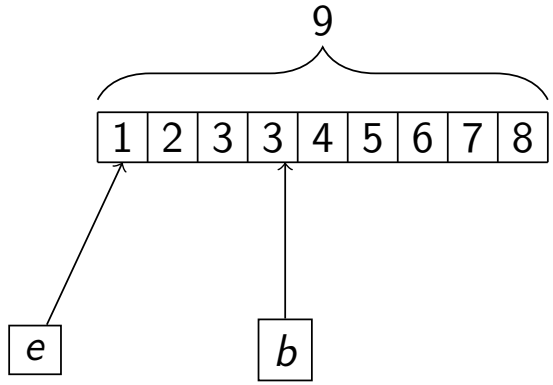
An array is a data structure consisting of elements of the same type.




```
1  int e[9];                /* defines an array of size 9; memory is not cleared! */
2  int* b;
3  int i;
4
5  for( i = 0; i < 9; ++i )
6    e[i] = i;              /* access value of i-th entry */
7
8  b = e;                   /* b points to the array */
9  b = &e[0];               /* nothing has changed */
10
11 for( i = 0; i < 4; ++i )
12 {
13   (*b) += 1;             /* increment value */
14   b += 1;                /* move pointer */
15 }
```

Arrays

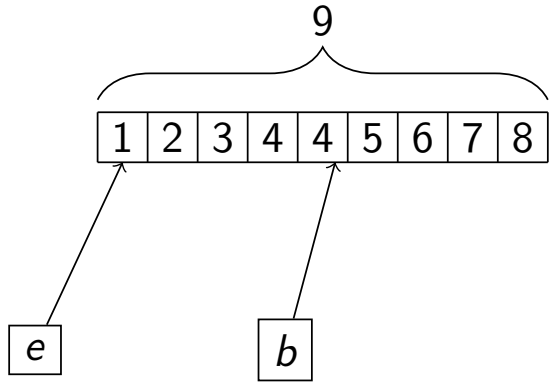
An array is a data structure consisting of elements of the same type.




```
1  int e[9];                /* defines an array of size 9; memory is not cleared! */
2  int* b;
3  int i;
4
5  for( i = 0; i < 9; ++i )
6    e[i] = i;              /* access value of i-th entry */
7
8  b = e;                   /* b points to the array */
9  b = &e[0];               /* nothing has changed */
10
11 for( i = 0; i < 4; ++i )
12 {
13   (*b) += 1;             /* increment value */
14   b += 1;                /* move pointer */
15 }
```

Arrays

An array is a data structure consisting of elements of the same type.




```
1  int e[9];                /* defines an array of size 9; memory is not cleared! */
2  int* b;
3  int i;
4
5  for( i = 0; i < 9; ++i )
6    e[i] = i;              /* access value of i-th entry */
7
8  b = e;                   /* b points to the array */
9  b = &e[0];               /* nothing has changed */
10
11 for( i = 0; i < 4; ++i )
12 {
13   (*b) += 1;             /* increment value */
14   b += 1;                /* move pointer */
15 }
```

Arrays

How to create an integer array of size n?

Arrays

How to create an integer array of size n?

WRONG:

```
1  int* createArray(int n)
2  {
3    int b[n];
4    return b;
5  }
```

- `int b[...];` does only work for constants
- memory for b is only allocated in its scope

SCIP's memory management

SCIP has its own memory management.

Use...

- `SCIPallocBufferArray(...)` to allocate memory for an array
- `SCIPfreeBufferArray(...)` to free allocated memory for an array
- `SCIPallocMemory(...)` to allocate memory for a struct
- `SCIPfreeMemory(...)` to free allocated memory for a struct

Note...

- wrappers for `malloc(...)` and `free(...)`
- memory is not cleared after allocation
- more functions to allocate memory in SCIP, see `src/scip/scip.h`

SCIP's memory management

Allocation functions in SCIP are macros but think of them as:

```
1  /* allocates memory for an array (only for temporary use) */
2  SCIP_RETCODE SCIPallocBufferArray(SCIP* scip, void** ptr, int size);
3                                          /* void* can store any kind of pointer */
4  /* frees a buffer array */
5  void SCIPfreeBufferArray(SCIP* scip, void** ptr);
6
7  /* allocates memory for a struct */
8  SCIP_RETCODE SCIPallocMemory(SCIP* scip, void** ptr) );
9
10 /* frees an allocated struct */
11 void SCIPfreeMemory(SCIP* scip, void** ptr);
```

SCIP's memory management

Allocation functions in SCIP are macros but think of them as:

```
1  /* allocates memory for an array (only for temporary use) */
2  SCIP_RETCODE SCIPallocBufferArray(SCIP* scip, void** ptr, int size);
3                                          /* void* can store any kind of pointer */
4  /* frees a buffer array */
5  void SCIPfreeBufferArray(SCIP* scip, void** ptr);
6
7  /* allocates memory for a struct */
8  SCIP_RETCODE SCIPallocMemory(SCIP* scip, void** ptr) );
9
10 /* frees an allocated struct */
11 void SCIPfreeMemory(SCIP* scip, void** ptr);
```

```
1  SCIP_RETCODE foo(SCIP* scip, int n)
2  {
3    SCIP_Real* b;
4
5    SCIP_CALL( SCIPallocBufferArray(scip, &b, n) ); /* array is not cleared */
6
7    ...;                                /* do something with the array */
8
9    SCIPfreeBufferArray(scip, &b);   /* important to free memory afterwards */
10
11   return SCIP_OKAY;
12 }
```

SCIP's memory management

How to allocate a two dimensional array

```
1  SCIP_RETCODE foo(SCIP* scip, int n, int m)
2  {
3    SCIP_Bool** b;          /* think of b as: array of arrays */
4    int i;
5
6    SCIP_CALL( SCIPallocBufferArray(scip, &b, n) ); /* as before */
7
8    for( i = 0; i < n; ++i )
9    {
10     /* this is an allocation of an array as we have seen it before,
11      * the type of b[i] is SCIP_Bool*
12      */
13     SCIP_CALL( SCIPallocBufferArray(scip, &b[i], m) );
14   }
15
16   b[0][0] = TRUE;         /* set a value of two dimensional array */
17
18   /* free memory in reversed order */
19   for( i = n - 1; i >= 0; --i )
20   {
21     SCIPfreeBufferArray(scip, &b[i])
22   }
23
24   /* free b at the end */
25   SCIPfreeBufferArray(scip, &b);
26 }
```

Refreshing C for SCIP Programming

Task

Solve the following IP for given $n, k \in \mathbb{N}$:

$$\min \sum_{i=1}^{n} -|\sin(i)| \cdot x_i$$

$$s.t. \sum_{i=1}^{n} i \cdot x_i \leq k$$

$$x \in \{0,1\}^n$$

```c
#include "scip/scip.h"
#include "scip/scipdefplugins.h"        /* to include default plugins */

/* adds variables and constraints to SCIP */
SCIP_RETCODE createIP(SCIP* scip, int n, int k){ ...; }

/* creates and solves the IP for given n and k */
SCIP_RETCODE solveIP(int n, int k)
{
  SCIP* scip;

  /* create and initialize SCIP data structures */
  SCIP_CALL( SCIPcreate(&scip) );

  /* include default plugins */
  SCIP_CALL( SCIPincludeDefaultPlugins(scip) );

  /* create the IP */
  SCIP_CALL( createIP(scip, n, k) );

  /* solve the IP */
  SCIP_CALL( SCIPsolve(scip) );

  /* print best solution */
  SCIP_CALL( SCIPprintBestSol(scip, NULL, FALSE) );

  /* frees SCIP data structure */
  SCIP_CALL( SCIPfree(&scip) );

  return SCIP_OKAY;
}
```

Create an IP with SCIP

```
1  /* adds variables and constraints to SCIP */
2  SCIP_RETCODE createIP(SCIP* scip, int n, int k)
3  {
4    /* check preconditions */
5    assert(scip != NULL);
6    assert(n > 0 && k > 0);
7
8    /* create empty problem */
9    SCIP_CALL( SCIPcreateProbBasic(scip, "coatwork-IP") );
10
11   /* 1.) create variables */
12   ...;
13
14   /* 2.) create constraint */
15   ...;
16
17   /* 3.) free memory */
18   ...;
19
20   return SCIP_OKAY;
21 }
```

1.) Create Variables

```
1   /* add this to the top of createIP(...) */
2   SCIP_VAR** vars;                        /* array of SCIP_VAR* pointer */
3   char varname[SCIP_MAXSTRLEN];           /* char array (string) */
4   int i;
5
6   /* allocate array which is only needed in this function */
7   SCIP_CALL( SCIPallocBufferArray(scip, &vars) );
8
9   for( i = 0; i < n; ++i )
10  {
11    /* store variable name in varname */
12    (void) SCIPsnprintf(varname, SCIP_MAXSTRLEN, "x_%d", i);
13
14    /* create variable */
15    SCIP_CALL( SCIPcreateVarBasic(scip, &vars[i], varname,
16          0.0, 1.0, -REALABS(sin(i+1)), SCIP_VARTYPE_BINARY) );
17
18    /* add variable to SCIP */
19    SCIP_CALL( SCIPaddVar(scip, vars[i]) );
20  }
```

2.) Create Constraints

```
1   /* add this to the top of createIP(...) */
2   SCIP_CONS* cons;
3   SCIP_Real* coefs;    /* array of SCIP_Real values */
4
5   /* allocate array for all coefficients */
6   SCIP_CALL( SCIPallocBufferArray(scip, &coefs, n) );
7
8   /* set correct coefficients */
9   for( i = 0; i < n; ++i )
10    coefs[i] = i + 1.0; /* (i+1) x_i */
11
12  /* create linear constraint */
13  SCIP_CALL( SCIPcreateConsBasicLinear(scip, &cons, "cons1",
14                n, vars, coefs, -SCIPinfinity(scip), k) );
15
16  /* add constraint to SCIP */
17  SCIP_CALL( SCIPaddCons(scip, cons) );
18
19  /* release constraint (if not needed anymore) */
20  SCIP_CALL( SCIPreleaseCons(scip, &cons) );
21
22  /* release array for coefficients */
23  SCIP_CALL( SCIPfreeBufferArray(scip, &coefs) );
```

3.) Free Memory

```
1  /* release variables (if not needed anymore) */
2  for( i = n - 1; i >= 0; --i )
3  {
4    /* release i-th variable */
5    SCIP_CALL( SCIPreleaseVar(scip, &vars[i]) );
6  }
7
8  /* free array for variables */
9  SCIP_CALL( SCIPfreeBufferArray(scip, &vars) );
```

Let us solve some IPs!

- find the complete source code here: projects/scipoptsuite-3.2.0/scip/examples/FirstIP/cmain.c
- compile with: `"make"` or `"make OPT=dbg"`
- usage: `./bin/firstmip <n> <k>`

Try to add the following things:

- display statistics after solving the problem
- print the solution value of the best solution in a debug message
- set a time and a node limit before calling `SCIPsolve(scip)`

Debug Output

Put #define SCIP_DEBUG on top of a file to activate debug output.

```c
/* write your own debug messages */
SCIPdebugMessage("a debug message printing a retcode %d\n", x);
```

```
presolved problem has 14 variables (0 bin, 9 int, 5 impl, 0 cont) and 12 constraints
      4 constraints of type <varbound>
      8 constraints of type <linear>
transformed objective value is always integral (scale: 30)
Presolving Time: 0.00

 time | node  | left  |LP iter|LP it/n| mem |mdpt |frac |vars |cons |cols |rows |cuts |confs|strbr|  dualbound   | primalbound  |  gap
T 0.0s|     1 |     0 |     0 |     - | 257k|   0 |   - |  14 |  12 |  14 |  12 |   0 |   0 |   0 |      --      | 1.202700e+06 |    Inf
  0.0s|     1 |     0 |     9 |     - | 254k|   0 |   8 |  14 |  12 |  14 |  12 |   0 |   0 |   0 | 1.167875e+06 | 1.202700e+06 |  2.98%
  0.0s|     1 |     0 |    11 |     - | 272k|   0 |   8 |  14 |  12 |  14 |  14 |   2 |   0 |   0 | 1.167905e+06 | 1.202700e+06 |  2.98%
  0.0s|     1 |     0 |    15 |     - | 287k|   0 |   8 |  14 |  12 |  14 |  17 |   5 |   0 |   0 | 1.167930e+06 | 1.202700e+06 |  2.98%
  0.0s|     1 |     0 |    18 |     - | 313k|   0 |   8 |  14 |  12 |  14 |  18 |   6 |   0 |   0 | 1.167945e+06 | 1.202700e+06 |  2.98%
  0.0s|     1 |     0 |    21 |     - | 337k|   0 |   8 |  14 |  12 |  14 |  20 |   8 |   0 |   0 | 1.167966e+06 | 1.202700e+06 |  2.97%
  0.0s|     1 |     0 |    24 |     - | 351k|   0 |   8 |  14 |  12 |  14 |  22 |  10 |   0 |   0 | 1.167986e+06 | 1.202700e+06 |  2.97%
  0.0s|     1 |     0 |    26 |     - | 370k|   0 |   8 |  14 |  12 |  14 |  23 |  11 |   0 |   0 | 1.167989e+06 | 1.202700e+06 |  2.97%
F 0.0s|     1 |     0 |    51 |     - | 373k|   0 |   0 |  14 |  12 |  14 |  23 |  11 |   0 |   0 | 1.167989e+06 | 1.201500e+06 |  2.87%
  0.0s|     1 |     0 |    51 |     - | 372k|   0 |   8 |  14 |  12 |  14 |  23 |  11 |   0 |   0 | 1.167989e+06 | 1.201500e+06 |  2.87%
  0.0s|     1 |     0 |    53 |     - | 390k|   0 |   8 |  14 |  12 |  14 |  24 |  12 |   0 |   0 | 1.168004e+06 | 1.201500e+06 |  2.87%
  0.0s|     1 |     0 |    62 |     - | 391k|   0 |   6 |  14 |  12 |  14 |  29 |  17 |   0 |   0 | 1.171134e+06 | 1.201500e+06 |  2.59%
  0.0s|     1 |     0 |    67 |     - | 391k|   0 |   4 |  14 |   7 |  14 |  29 |  17 |   0 |   6 | 1.183158e+06 | 1.201500e+06 |  1.55%
  0.0s|     1 |     0 |    68 |     - | 399k|   0 |   4 |  14 |   7 |  14 |  30 |  18 |   0 |   6 | 1.183169e+06 | 1.201500e+06 |  1.55%
  0.0s|     1 |     0 |    70 |     - | 399k|   0 |   2 |  14 |   7 |  14 |  31 |  19 |   0 |   6 | 1.184300e+06 | 1.201500e+06 |  1.45%
 time | node  | left  |LP iter|LP it/n| mem |mdpt |frac |vars |cons |cols |rows |cuts |confs|strbr|  dualbound   | primalbound  |  gap
  0.0s|     1 |     0 |    70 |     - | 398k|   0 |   2 |  14 |   7 |  14 |  20 |  19 |   0 |   6 | 1.184300e+06 | 1.201500e+06 |  1.45%
  0.0s|     1 |     2 |    70 |     - | 398k|   0 |   2 |  14 |   7 |  14 |  20 |  19 |   0 |   6 | 1.184300e+06 | 1.201500e+06 |  1.45%
```

Debug Output

Put #define SCIP_DEBUG on top of a file to activate debug output.

```c
/* write your own debug messages */
SCIPdebugMessage("a debug message printing a retcode %d\n", x);
```

```
presolved problem has 14 variables (0 bin, 9 int, 5 impl, 0 cont) and 12 constraints
      4 constraints of type <varbound>
      8 constraints of type <linear>
transformed objective value is always integral (scale: 30)
Presolving Time: 0.00

 time | node  | left  |LP iter|LP it/n| mem |mdpt |frac |vars |cons |cols |rows |cuts |confs|strbr|  dualbound   | primalbound  |  gap
T 0.0s|     1 |     0 |     0 |     - | 257k|   0 |   - |  14 |  12 |  14 |  12 |   0 |   0 |   0 |      --      | 1.202700e+06 |    Inf
  0.0s|     1 |     0 |     9 |     - | 254k|   0 |   8 |  14 |  12 |  14 |  12 |   0 |   0 |   0 | 1.167875e+06 | 1.202700e+06 |   2.98%
[src/scip/heur_rounding.c:577] debug: executing rounding heuristic: 12 LP rows, 8 fractionals
[src/scip/heur_rounding.c:634] debug: rounding heuristic: nfrac=8, nviolrows=0, obj=1.16788e+06 (best possible obj: 1.1604e+06)
[src/scip/heur_rounding.c:671] debug: rounding heuristic: search rounding variable and try to stay feasible
[src/scip/heur_rounding.c:682] debug: rounding heuristic:  -> round var <t_STM4>, oldval=69.6181, newval=70, obj=90
[src/scip/heur_rounding.c:700] debug: rounding heuristic:  -> nfrac=7, nviolrows=2, obj=1.16891e+06 (best possible obj: 1.1631e+06)
[src/scip/heur_rounding.c:634] debug: rounding heuristic: nfrac=7, nviolrows=2, obj=1.16891e+06 (best possible obj: 1.1631e+06)
[src/scip/heur_rounding.c:655] debug: rounding heuristic: try to fix violated row <ANZ4>: 0 <= -3.81944 <= -0
[src/scip/heur_rounding.c:682] debug: rounding heuristic:  -> round var <t_ANM3>, oldval=13.5113, newval=14, obj=50
[src/scip/heur_rounding.c:700] debug: rounding heuristic:  -> nfrac=6, nviolrows=3, obj=1.16964e+06 (best possible obj: 1.1646e+06)
[src/scip/heur_rounding.c:634] debug: rounding heuristic: nfrac=6, nviolrows=3, obj=1.16964e+06 (best possible obj: 1.1646e+06)
[src/scip/heur_rounding.c:655] debug: rounding heuristic: try to fix violated row <STD3>: 8000 <= 7951.13 <= 1e+20
[src/scip/heur_rounding.c:682] debug: rounding heuristic:  -> round var <t_STM3>, oldval=62.3409, newval=63, obj=90
[src/scip/heur_rounding.c:700] debug: rounding heuristic:  -> nfrac=5, nviolrows=3, obj=1.17142e+06 (best possible obj: 1.1673e+06)
[src/scip/heur_rounding.c:634] debug: rounding heuristic: nfrac=5, nviolrows=3, obj=1.17142e+06 (best possible obj: 1.1673e+06)
[src/scip/heur_rounding.c:655] debug: rounding heuristic: try to fix violated row <ANZ3>: 486 <= 492.591 <= 486
[src/scip/heur_rounding.c:682] debug: rounding heuristic:  -> round var <t_ANM2>, oldval=5.21304, newval=6, obj=50
[src/scip/heur_rounding.c:700] debug: rounding heuristic:  -> nfrac=4, nviolrows=4, obj=1.1726e+06 (best possible obj: 1.1688e+06)
[src/scip/heur_rounding.c:634] debug: rounding heuristic: nfrac=4, nviolrows=4, obj=1.1726e+06 (best possible obj: 1.1688e+06)
[src/scip/heur_rounding.c:655] debug: rounding heuristic: try to fix violated row <STD2>: 900 <= 821.304 <= 1e+20
[src/scip/heur_rounding.c:682] debug: rounding heuristic:  -> round var <t_ANM1>, oldval=9.47536, newval=10, obj=140
[src/scip/heur_rounding.c:700] debug: rounding heuristic:  -> nfrac=3, nviolrows=3, obj=1.1748e+06 (best possible obj: 1.173e+06)
[src/scip/heur_rounding.c:634] debug: rounding heuristic: nfrac=3, nviolrows=3, obj=1.1748e+06 (best possible obj: 1.173e+06)
[src/scip/heur_rounding.c:655] debug: rounding heuristic: try to fix violated row <ANZ3>: 486 <= 480 <= 486
[src/scip/heur_rounding.c:678] debug: rounding heuristic:  -> didn't find a rounding variable
  0.0s|     1 |     0 |    11 |     - | 272k|   0 |   8 |  14 |  12 |  14 |  14 |   2 |   0 |   0 | 1.167905e+06 | 1.202700e+06 |   2.98%
```

SCIP Code Structure

How to find the functions to use for your project:

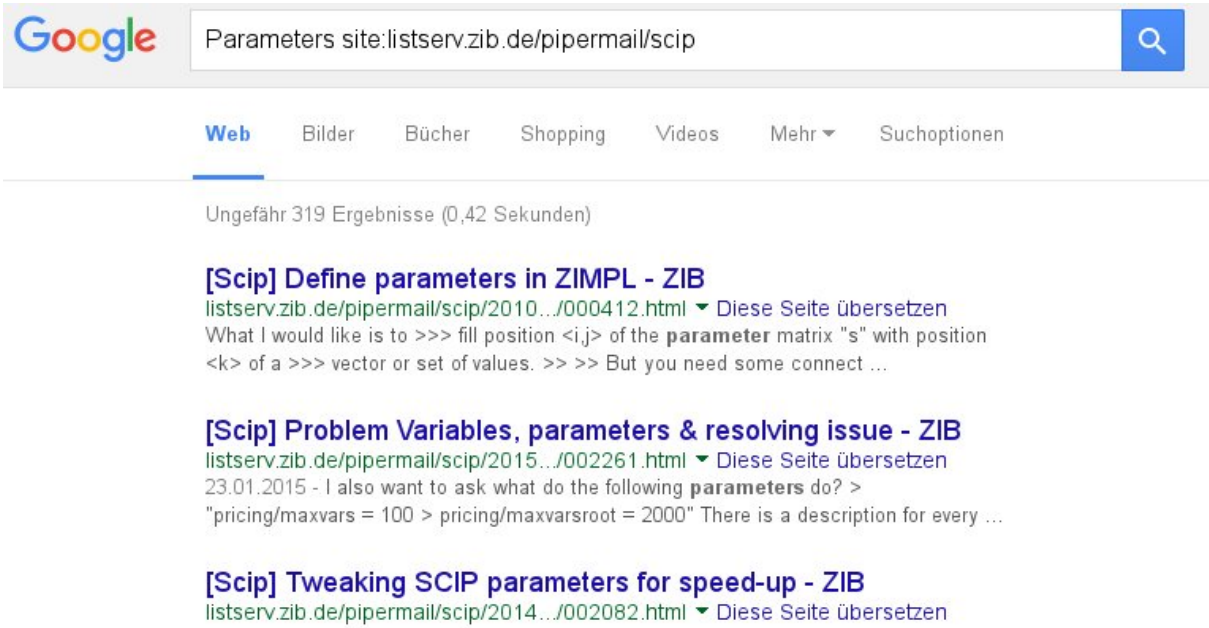
- either in `src/scip/scip.h`
- or in one of `src/scip/pub_<name>.h`

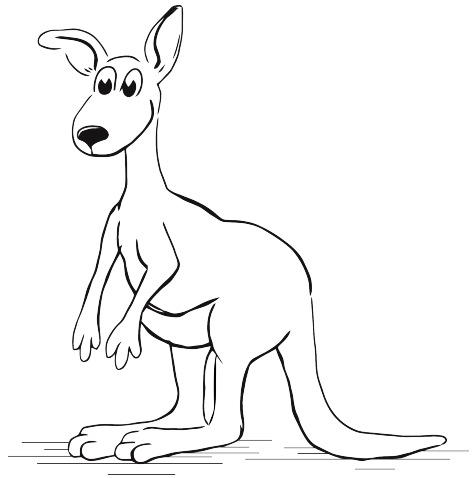
How to create and add constraints of various types:

- `#include scip/cons_<name>.h`
- then use `SCIPcreateConsBasic<name>(...)`
- or `SCIPcreateCons<name>(...)` (more customizable)

More SCIP infos

- documentation, coding examples, and FAQ:
  `http://scip.zib.de/doc/html`
- SCIP mailing list archives:
  `http://listserv.zib.de/pipermail/scip/`
- or use Google search with option
  `site:listserv.zib.de/pipermail/scip`





Questions?