

Refreshing C for SCIP programmers

CO@Work Berlin

Thorsten Koch

MATHEON/ZIB

22.10.2009



Berlin
Mathematical
School



DFG Research Center **MATHEON**
Mathematics for key technologies



Hello World

```
1 #include <stdio.h> /* Include predefined stuff */
2
3 /* This is a comment.
4  * Every C program has exactly one "main" function.
5  */
6 int main()
7 {
8     puts(" Hello _World" );
9
10    return 0;
11 }
```

C language constructs I

```
1 #include <stdio.h>           /* <name> for system stuff */
2 #include <assert.h>
3 #include "c_one.h"          /* "name" for our own stuff */
4
5 double sumup(int a, int b, char* msg) /* not a real function */
6 {
7     int i;                   /* Declare variables */
8     double sum = 0.0;        /* Variables can be initialised */
9
10    assert(a >= 0 && b >= 0); /* assert to state preconditons */
11    assert(msg != NULL);     /* Pointers might point nowhere */
12
13    printf("%s %d %d\n", msg, a, b); /* Output specifiers */
14
15    if (b < a) {              /* Conditional */
16        int t = a; a = b; b = t; /* Again a block start */
17    }
18    assert(a <= b);
19
20    for(i = a; i < b; i++)    /* Loop. Notice <, ++, += */
21        sum += i;           /* No block start needed */
22
23    return sum;
24 }
```

C language constructs II

c_one.h

```
1 #ifndef _C_ONE_H_
2 #define _C_ONE_H_
3
4 extern double sumup(int a, int b, char* message);
5
6 #endif /* _C_ONE_H_ */
```

c_main.c

```
1 #include <stdio.h> /* Include predefined stuff */
2 #include <stdlib.h>
3 #include <assert.h>
4
5 #include "c_one.h"
6
7 int main() /* There has to be exactly one "main" */
8 {
9     printf("Result = %g\n", sumup(2, 4, "calling _sumup"));
10
11     return EXIT_SUCCESS;
12 }
```

C language constructs III

```
1  #define SPECIAL_K    17          /* Define a constant */
2  #define SPECIAL_X    42
3
4  /* Function without return value
5   */
6  void strange(int x)
7  {
8      switch(x)          /* Case disjunction. Simple numbers only */
9      {
10     case 0 :
11         printf("Zero\n");
12         break;          /* Important, otherwise fallthrough */
13     case SPECIAL_K :   /* Preprocessor substitutes with 17 */
14         printf("Special-K");
15         break;
16     case SPECIAL_X :
17         printf("Special-X");
18         /*FALLTHROUGH*/
19     default:          /* Catch the rest */
20         printf("Special-%d\n", x);
21         break;
22     }
23 }
```

Compounds

```
1 #include <stdio.h>
2
3 struct data                               /* Declare a compound */
4 {
5     int a;
6     int b;
7 };
8
9 typedef struct data Data;                 /* Add a shortcut name */
10
11 int main()
12 {
13     Data d[4] = {{0,1},{2,3},{4,5},{6,7}};
14     int i;
15
16     d[1].a = 17;
17     d[2].b = -5;
18
19     for(i = 0; i < sizeof(d) / sizeof(d[0]); i++)
20         printf("Elem %d, a=%2d b=%+2d\n", i, d[i].a, d[i].b);
21
22     return 0;
23 }
```

Operators and types, etc.

```
1 <      less than
2 >      greater than
3 <=     less equal
4 >=     greater equal
5 ==     equal
6 !=     not equal
7 !      logic not
8 &&     logic and
9 ||     logic or
10 |     bit or
11 &     bit and
12 ^     bit xor
13 ~     bit negation
14 <<    bit shift left
15 >>    bit shift right
16 %     modulo
17 =     assignment
```

```
18
19 unsigned / signed
20 char / short / int / long / long long
21 float / double
```

no power operator, e.g. $2^{18} = 16$

no true boolean, no string type

```
if uses shortcut evaluation
if (f(7) > 5 && g(2) != 0)
g might not be called
be aware, f and g might have
side effects
```

```
x = f() + g() * h()
order of evaluation is not
defined
```

Language Standards

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int c =
6     10 /*
7     /*/ 5
8     ;
9     printf("%d\n", c);
10    return 0;
11 }
```

Language Standards

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int c =
6     10 /*
7     /*/ 5
8     ;
9     printf("%d\n", c);
10    return 0;
11 }
```

```
$ gcc -std=c89 test.c
```

```
$ ./a.out
```

```
2
```

```
$ gcc -std=c99 test.c
```

```
$ ./a.out
```

```
10
```

Pointer and memory allocation

```
1 #include <stdlib.h>
2 #include <assert.h>
3
4 #define DSIZE 1000
5
6 void initialize(size_t size, double *e)
7 {
8     size_t i;
9     assert(e != NULL);
10    for(i = 0; i < size; i++)
11        e[i] = i * 5.7;
12 }
13 int main()
14 {
15     double* d = calloc(DSIZE, sizeof(*d)); /* Allocate memory */
16
17     if (d == NULL) /* Was it successful? */
18         return EXIT_FAILURE;
19
20     initialize(DSIZE, d);
21     free(d); /* Release the memory */
22     return EXIT_SUCCESS;
23 }
```

Pointers and arrays

```
1 char a[100];      /* defines an array      */
2 char* p = a;     /* p points to the array */
3 char* q = &a[0]; /* a == &a[0]          */
4
5 a[1] = 'A';
6 p[1] = 'B';
7 *(p+1) = 'C';
8 *++q = 'D'; /* q changes also */
9 *(a+1) = 'E';
10 *++a = 'F'; /* illegal */
11
12 p = "Hi_there";
13 strcpy(a, p);
14
15 if (p == q) /* compares addresses, not strings */
16
17 if (!strcmp(a, p)) /* compares strings */
```

Today it will not get worse than this

```
1  int max(int a, int b)
2  {
3      return a < b ? b : a;
4  }
5
6  void copystring(char *t, const char* s)
7  {
8      while(*t++ = *s++);
9  }
10
11 void bar(int** p, size_t size)
12 {
13     *p = malloc(size, sizeof(**p));
14 }
15
16 void foo()
17 {
18     int* q = NULL;
19
20     bar(&q, 1000);
21
22     q[4] = 5;
23 }
```

Fun with FP Arithmetic

```
1 double s[2048];
2 double e = 1;
3 int    n = 0;
4
5 do
6 {
7     n    = n + 1;
8     e    = e / 2;
9     s[n] = 1 + e;
10 }
11
12 while(    e > 0); /* alternative 1 */
13 while(1 + e > 1); /* alternative 2 */
14 while(s[n] > 1); /* alternative 3 */
```

Does the loop terminate?

Will the program crash ?

If it terminates, will n have the same value in all alternatives?

(Lines 7,8,9)

Fun with FP Arithmetic

```
1 double s[2048];
2 double e = 1;
3 int    n = 0;
4
5 do
6 {
7     n    = n + 1;
8     e    = e / 2;
9     s[n] = 1 + e;
10 }
11
12 while(    e > 0); /* n = 1075 */
13 while(1 + e > 1); /* n = 64   (Intel P4) */
14 while(s[n] > 1); /* n = 53   */
```

Does the loop terminate? Yes.

Will the program crash ? No.

If it terminates, will n have the same value in all alternatives?

No. n is different in many cases, possibly depending on the architecture and the compiler (settings).

Floating point values should not be compared directly against each other.

```
1 SCIP_Real double;
2
3 /* returns value treated as infinity */
4 SCIP_Real SCIPinfinity(SCIP* scip);
5
6 /* compare value against infinity */
7 SCIP_Bool SCIPisInfinity(SCIP* scip , SCIP_Real val);
8
9 /* checks, if value is in range epsilon of 0.0 */
10 SCIP_Bool SCIPisZero(SCIP* scip , SCIP_Real val);
11
12 /* checks, if values are in range of feasibility tolerance */
13 SCIP_Bool SCIPisFeasEQ(SCIP* scip , SCIP_Real val1 , SCIP_Real val2 );
14
15     SCIP_Real a;
16     SCIP_Real b;
17
18     if (SCIPisFeasEQ(scip , a , b))
19         printf("Equal up to a tolerance\n");
```

SCIP return codes

```
1  SCIP_RETCODE          /* return code for SCIP methods */
2
3  /* Some of the possible values */
4  SCIP_OKAY             /* normal termination */
5  SCIP_ERROR           /* unspecified error */
6  SCIP_NOMEMORY        /* insufficient memory error */
7  SCIP_READERROR       /* file read error */
8  SCIP_WRITEERROR      /* file write error */
9  SCIP_LPERROR         /* error in LP solver */
10 SCIP_NOPROBLEM       /* no problem exists */
11 SCIP_INVALIDCALL     /* method cannot be called at this time */
12 SCIP_INVALIDRESULT   /* method returned an invalid result code */
13 SCIP_PARAMETERUNKNOWN /* the parameter not found */
14
15 /* The easy way to check */
16 SCIP_RETCODE some_function()
17 {
18     ... ;
19     SCIP_CALL(SCIPAllocBufferArray(...));
20     ... ;
21
22     return SCIP_OKAY;
23 }
```

Calling SCIP functions

```
1 SCIP_RETCODE some_function()
2 {
3     /* Check return value, if not SCIP_OKAY then
4      * complain and return the bad news */
5     SCIP_CALL(SCIPAllocBufferArray(...));
6
7     return SCIP_OKAY;
8 }
9 SCIP_RETCODE some_function()
10 {
11     /* Check return value, if not SCIP_OKAY then
12      * immediately pass return code on */
13     SCIP_CALL_QUIET(SCIPAllocBufferArray(...));
14
15     return SCIP_OKAY;
16 }
17 void some_function()
18 {
19     /* Check return value, if not SCIP_OKAY then
20      * scream and die */
21     SCIP_CALL_ABORT(SCIPAllocBufferArray(...));
22 }
```

Some details

```
1 #include "scip/scip.h"
2
3 SCIP_Bool      TRUE / FALSE
4 SCIP_Longint   long long
5
6 #define SCIP_DEBUG
7 {
8     SCIPdebugMessage(" Did not work %d\n", retcode );
9     ...;
10 }
```

SCIP has its own memory management

```
1 SCIP_RETCODE fun_that_needs_a_buffer(SCIP* scip, int max_elems)
2 {
3     int* store_here;
4     int i;
5
6     SCIP_CALL(SCIPAllocBufferArray(scip, &store_here, max_elems));
7
8     for(i = 0; i < max_elems; i++)
9         store_here[i] = 17;
10
11     ...; /* use it */
12
13     SCIPfreeBufferArray(scip, &store_here);
14 }
```

```
1 class Test // abstract base class
2 {
3 private:
4     double b;
5 protected:
6     int c;
7 public:
8     Test() : c(0) {};
9     virtual int check() = 0;
10 };
11
12 class RealTest : Test
13 {
14 public:
15     RealTest() : Test() {}
16     int check() { return c != 0; }
17 };
18
19 int main()
20 {
21     RealTest t;
22
23     return t.check() ? 1 : 13;
24 }
```

Questions?

Refreshing C for SCIP programmers

CO@Work Berlin

Thorsten Koch

MATHEON/ZIB

22.10.2009



Berlin
Mathematical
School



DFG Research Center MATHEON
Mathematics for key technologies

