Introduction to Programming

Block Tutorial C/C++

Michael Bader

Master's Program
"Computational Science and Engineering"

C/C++ Tutorial – Overview

- From Maple to C
- Variables, Operators, Statements
- Functions: declaration, definition, parameters
- Arrays and Pointers
- From Pointers to Dynamical Data Structures
- Object Oriented Programming: From C to C++

Monday Lesson – Overview

- Maple and C: the differences
- Compiling
- First Steps in C: "Hello World!"
- Variables in C: data types, declaration and assignments
- C operators
- C statements
- basic input and output

Maple and C – Differences

Maple's programming language:

- needs Maple as runtime environment
- is interactive: user calls functions, Maple prints the results
- is declaration-based: procedures can be (re-)defined any time, the last definition is used

C programming language:

- needs "compiler" to translate source code to executable program
- is not interactive. Input/output has to be programmed explicitly.
- is imperative: sequence of instructions and data flow of a program is predetermined. Changing program code requires recompilation.

Maple vs. C – Variables

Maple:

Variables hold expressions

- mathematical expressions
- any type of expression
- type of expression may change
- logical data structures: lists, sets, tables

C:

Variables are cells of memory

- contain certain type of variable (integer, floating point, . . .)
- type of variable is fixed and declared in advance
- physical data structures: structs, arrays

$\Rightarrow$ **Fundamentally different concept of variables!**

Maple and C – Similarities

Binomial Coefficients in Maple and C – Recursive Example:

in Maple:

```
binom := proc(n, k)
   if k=0 or k=n
   then return 1
   else
      return binom(n-1,k-1)
           + binom(n-1,k)
   end if
end proc;
```

in C:

```
int binom(int n, int k) {
   if ( k==0 || k==n )
      return 1;
   else
      return binom(n-1,k-1)
           + binom(n-1,k) ;
}
```

Maple and C – Similarities

Binomial Coefficients in Maple and C – using loops:

in Maple:

```
binom := proc(n, k)
   local i,nom,den;
   nom = 1; den = 1;
   for i from 1 to k do
      nom := nom*(n+1-i);
      den := den*i;
   end do;
   return nom/den;
end proc;
```

in C:

```
int binom(int n, int k) {
   int i;
   float den=1, nom=1;
   for(i=1;i<=k;i++) {
      nom = nom*(n+1-i);
      den = den*i;
   }
   return nom/den ;
}
```

Maple and C – Compiling

Maple

- type procedure in worksheet
- press return

  → Maple checks syntax
  → Maple builds internal representation of procedure
- call procedure

  → Maple computes (and prints) result

C

- type program using text editor
- start C compiler

  → Compiler checks syntax
  → Compiler builds executable program (machine code)
- start generated executable

  → code does whatever it was designed to do

First Steps in C – Compiling

I. Before compiling: type code into a text file $\rightarrow$ `prog.c`

II. Start compiler: `gcc prog.c`

$\rightarrow$ check syntax
$\rightarrow$ check declarations (all variables declared? all assignments type-correct?)
$\rightarrow$ generate (meta-)code
$\rightarrow$ optimize
$\rightarrow$ generate "object code"
$\rightarrow$ "link" object code (add libraries, other code files, etc.), leads to executable program

III. Start program

First Steps in C – "Hello World"

```
#include <stdio.h>
main() {
   printf("Hello World\n");
}
```

`#include <stdio.h>`: include library for standard input/output operations

- `#include`: call to "C preprocessor", include another file
- `<stdio.h>`: *header file*, contains declarations for a standardized set ("library") of input/output functions.
- <> hint that the library is "provided by C".

First Steps in C – "Hello World"

```
#include <stdio.h>
main() {
   printf("Hello World\n");
}
```

`main()` `...` : define a function called `main`

- Question: Where does a C program start?

  – At the first line?
  – What if there is more than on file containing program code?
  → all programs start with a call to `main()`

- {} begin/end the definition of the function `main`

First Steps in C – "Hello World"

```
#include <stdio.h>
main() {
   printf("Hello World\n");
}
```

`printf()`: generate output

- `printf()` is a function defined in `stdio.h`
- prints its parameters to "standard output" (i.e. screen/terminal/...)

First Steps in C – "Hello World"

```
#include <stdio.h>
main() {
   printf("Hello World\n");
}
```

`"Hello World!\n"`: a character string

- just like in Maple
- `\n` represents a linefeed (non-printable character)

$\Rightarrow$ Program prints string `"Hello World"` to standard output.

Variables in C

Basically, variables in C are just cells of memory.

⇒ we have to declare them first, i.e. tell the compiler what kind of variable they are

⇒ there are different types of variables.

⇒ different types of variables cannot hold the same type of information; a certain type of variable can only hold a certain type of information.

⇒ variables contain values; they do not contain complex terms or expressions

⇒ variables represent machine oriented data structures (i.e. sequences of cells in memory)

⇒ logical data structures (lists, sets, matrices) have to be implemented by the programmer

Basic C Syntax – Variables

Variables have to be declared first

- **always!**
- at the beginning of a function/block of code

Syntax: `<type> <variable>;`

Examples:

```
{ int i;
  float e,pi;
  char c;
}
```

Variables – Simple Types

Standard types:

`char`	characters and small integers
`int`	integers
`float`	floating point numbers

Additional types:

`double`	floating point numbers (double precision)
`long double`	floating point numbers (long precision)
`short [int]`	small integers
`long [int]`	big integers
`unsigned [int]`	unsigned integers
`unsigned char`	unsigned characters/small integers
`unsigned short`	unsigned small integers
`unsigned long`	unsigned big integers

Variables – Initialisation

Declare **and initialise** a variable with a certain value.

Examples:

```
{ int i=0;
  float e=2.7182, pi=3.14159;
  char c='A';
}
```

- avoids using a variable that has no value
- value should make sense, however

$\Rightarrow$ **good practice!**

Variables – Assignments

Assignments change the value of a certain variable:

```
{ i = 3;
  e = 2.7182818;
  pi = 3.14159265;
  c = 'B';
}
```

- assigned value **must** have the correct type
- notice the simple '='; there is no operator ':=' in C
- the semicolon finishes the assignment

Arithmetic Operators

Examples:

```
{ int i; float e,pi;
  i = 3 + 7;
  e = (2.7182 * 3.0) - 1.5;
  pi *= 2.0;
  i = 'Z' - 'G';
  i = ( e - pi ) / i;
}
```

- in principle, all operations have to be type correct!
- in practice, C tries very hard to adjust the type ("casting")
- use of parentheses like in Maple

Arithmetic Operators

+, −	summation and subtraction
*	multiplication
/	division; for integers, an integer division is performed
%	modulo operator, i.e. the remainder

Operator precedence:

- *,/,% are evaluated before +,− (like expected)
- unary plus/minus precedes *,/,%
- parentheses override precedence

Changing the Types of Variables

Sometimes, it is necessary to change the type of a variable.

Example:

```
int a=3, b=5;
double f;
f = a/b;
```

In this example, an integer division is performed. The result will be 0.

However, this does not seem to be the intended result.

$\Rightarrow$ we need a mechanism to change the type of a variable

Casting

Syntax: `(<type>) <expression>`

Example:

```
int a=3, b=5;
double f;
f = (double) a/ (double) b;
```

In this example, the result will be `0.6`!

C does automatic casting in a wide range of situations:

- expressions that mix `int` and `double` variables
- expressions that mix `float` and `double` variables

In most cases, C gets it right. There might be certain surprises, however ...

Statements

Overview:

- Assignment statements
- Conditional statements: `if ... else`, `switch ... case`
- block statements: `{ ... }`
- while statements: `while`, `do ... while`
- for statements: `for`
- other statements: `continue`, `break`

Assignment Statements

Examples:

```
i = j = 0 ;
e += 3.71 ;
i--; --i; i++; ++i;
```

Assignment Operators:

`=`	an assignment is an operator
`+=, -=, *=, /=, %=`	`i+=3` equiv. to `i=i+3`, etc.
`++`	increment operator
`--`	decrement operator
`i++, i--`	increment/decrement `i` after evaluation
`++i, --i`	increment/decrement `i` before evaluation

Conditional Statements

Syntax:

- `if ( <expression> ) <statement>;`
- `if ( <expression> ) <statement> else <statement>;`

Examples:

```
if ( a != 0 ) b /= a;
if ( x >= 0 ) abs = x; else abs = -x;

if ( x > -1 && x < 1 )
{  if ( x < 0 ) erg = 1+x; else erg = 1-x; }
else
   erg = 0;
```

Combining Statements – The Block Statement

Syntax: { `<statement> <statement> <statement> ...` }

Example:

```
{ int a=2,b=0;
  { int b=5;
    b = a*b;
  }
  printf("%d%\n",b);
}
```

- Note: the semicolon does not separate statements, it is part of certain statements!
- Are block statements useful? $\Rightarrow$ `if`-statements!

Conditional Statements Revisited – Boolean Expressions

Examples:

```
if ( a != 0 ) b /= a;
if ( x >= 0 ) abs = x; else abs = -x;
if ( x < -1 || x > 1 ) erg = 0;
if ( !a ) b /= a;
```

Boolean Expressions in C?

- there is no special type for boolean expressions!
- a "boolean" expression is considered false if it evaluates to zero; otherwise it is considered true.
- one should not exploit this too extensively . . .

"Boolean" Expressions – Overview

`==`, `!=`	is equal to/is not equal to
`<`, `<=`	less than (or equal)
`>`, `>=`	greater than (or equal)
`&&`	logical AND
`||`	logical OR
`!`	logical NOT (unary operator)

Operator precedence:

- `&&` precedes `||`
- `!=` and `==` precede `&&` and `||`
- `!` precedes `!=`, `==`, `&&`, and `||`
- parentheses override precedence – use them wisely!

While Statement

Syntax: `while (<expression>) <statement>`

Examples:

```
while (a >= b) a -= b ;
i = 1;
while (a >= 1) {
   a /= 2;
   i++;
}
```

- repeat executing `<statement>` until `<expression>` is false
- `<statement>` might not be executed at all

Do-While Statement

Syntax: `do <statement> while (<expression>);`

Example:

```
a = 1.0;
do {
   a *= 2.0;
   i--;
} while(i>0);
```

- repeat executing `<statement>` until `<expression>` is false
- `<statement>` is executed at least once!

For Statements

Syntax:

```
for(<expression>; <expression>; <expression>) <statement>
```

Examples:

```
for(i=1;i<=10;i++) a *= 2.0;

for(i=1;a>=1;i++) a /= 2.0;
```

- 1st expression: initialisation
- 2nd expression: stopping criterion
- 3rd expression: loop counter

For vs. While

```
for(<expression1>; <expression2>; <expression3>)
   <statement>
```

is equivalent to

```
<expression1>;
while(<expression2>) {
   <statement>
   <expression3>;
}
```

Note:

- an expression becomes a statement when followed by a semicolon!
- an assignment is also an expression!

For Statement – Examples

Binomial Coefficients:

```
coeff = 1;
for(i=n;i>=n-k+1;i--) coeff *= i;
for(i=1;i<=k;i++) coeff /= i;
```

Or, as a single loop:

```
nom = denom = 1;
for(i=1;i<=k;i++) {
   nom *= (n+1-i);
   denom *= i;
}
coeff = nom/denom;
```

Nested Statements

Example: prime numbers

```
for(i=3;i<1000;i+=2)
   if ( i%3 !=0 && i%5 !=0 && i%7 != 0)
      printf("%d might be a prime number\n",i);
for(i=2;i<1000;i++) {
   int dividers=0;
   for (j=2;j<i;j++)
      if (i%j == 0) {
         printf("%d is multiple of %d\n",i,j);
         dividers++; break;
      };
   if (dividers==0)
      printf("%d is prime!\n",i);
};
```

Input and Output

In contrast to Maple, we have to handle all user input/output ourselves. However, C provides some powerful library functions for this task:

- `printf` for output
- `scanf` for input

These are defined in the library `<stdio.h>`.

For today, we'll imagine we already know everything about functions . . .

Fortunately, **calling** functions in C and Maple looks exactly the same.

Actually, Maple adopted `printf` and `scanf` from C almost without any changes.

printf

Syntax: `printf(<string>, ...);`

`printf` requires a so-called *control string* and, possibly, further parameters.
The number of parameters depends on the *control string*!

Examples:

```
printf("Hello World\n");
printf("%d plus %d equals %d \n",a,b,(a+b));
printf("The logarithm of %d is %f\n",a,log(a));
```

The %-expressions are called *format specifiers*:

- for each format specifier, printf expects a parameter.
- specifier is replaced by formatted output of the parameter's value.

printf – Format Specifiers

specifier	print as:
`%d`, `%i`	decimal integer
`%f`	fixed point decimal number
`%e`, `%E`	floating point decimal number ("scientific" notation)
`%g`, `%G`	`%f` or `%e`,`%E` (shorter representation)
`%c`	character
`%s`	string

The format specifiers also determine the type of the parameters:

- `%d`, `%i` require `int` parameters;
- `%e`, `%f`, `%g` require `double` parameters; etc.

scanf

Syntax: `scanf(<string>, ...);`

`scanf` works very much like `printf`.
For each format specifier in the control string, `scanf` expects a parameter.

Examples:

```
printf("Please input a and b\n");
scanf("%d %f",&a,&b);
```

The `&`-operator is **strictly required**:

- it indicates that `a` and `b` are modified by scanf
- it works in a similar way as '`::evaln`' works in Maple

scanf – Format Specifiers

specifier	input type
%d, %i	int
%f, %e, %g	float
%lf, %le, %lg	double
%c	character
%s	string

- %d, %i require int parameters;
- notice the different specifiers for float and double:
  → %e, %f, %g require float parameters;
  → %le, %lf, %lg require double parameters.

scanf – Formatted Input

Example:

```
printf("Please input current date (dd / mm / yyyy)\n");
scanf("%d / %d / %d",&day,&month,&year);
```

- "white space" characters (space, tab, etc.) in the control string are ignored.
- other characters have to appear in the input in exactly the same way!

Input Source: "standard input"

- *input stream* (provided by the operating system)
- in most cases: input from the user (keyboard)
- can be redirected (on OS-level): e.g. input from a file

Last Example – What Does this Program Do?

```
main() {
   int i, k=1;
   double x, e=1.0, old=0.0;
   scanf("%lf",&x);
   while (old < e) {
      double f=1.0, p=1.0;
      for(i=1;i<=k;i++) {
         f *= (double) i;
         p *= x;
      }
      k++; old = e; e += p/f;
   }
   printf("%.16g\n",e);
}
```