Applied Programming and Computer Science, DD2325/appcs15

PODF, Programmering och datalogi för fysiker, DA7011

Autumn 2015

Lecture 3, Simple C programming

M. Eriksson
(with contributions from A. Maki and C. Edlund)

A (simple) generic C program

```
preprocessor directives  /* start with '#' end without ';' */

return-type function_1(parameters) {
  declarations
  statements
}

/*    ...
      ...
*/
return-type function_n(parameters) {
  declarations
  statements
}

int main(int argc, char **argv) {
  declarations
  statements
}
```

Recursive functions (ex. the factorial)

$$f(n) = \begin{cases} 1 & n = 1 \\ n \times f(n-1) & n > 1 \end{cases}$$

```
#include <stdio.h>

int factorial(int x) {
  if (x>1) {
      return x*factorial(x-1);
    }
  else
    return (1);
}

main(){
  int n;
  printf("Enter n: ");
  scanf("%d", &n);
  printf("The factorial of n is: %2d \n", factorial(n));
}
```

Statements

The major part of a C program consists of statements. A statement is a command which is executed when the program runs.

All statements should be terminated by a semi-colon, e.g.

```
   area = pi * radius * radius;
```

A statement can span several lines,

```
   area = pi *
          radius * radius;
```

Several statements can be collected in a compound statement which is treated as one statement by the compiler.
This is accomplished by braces,

```
{
   area = pi * radius * radius;
   circumference = 2 * pi * radius;
}
```

Basic types

All variables in C must have a type, which specifies what kind of data it will hold. A variable must be declared before use.

Basic types:

Type	**Description**
int	Integer values
float	Floating point numbers
double	Floating point numbers (higher accuracy, larger numbers)
char	Character

(There are different variants of the basic types, e.g. `unsigned int` which can only hold positive integers, and `long int` which can hold larger integers.)

In addition to the basic types C allows the user to specify his/her own data types.

Declaration of variables

Each program/function should begin with a declaration of variables. For example

```
main() {
  int   i, j;
  float f;

  /* statements */
}
```

This declaration defines two integers, `i` and `j`, and a floating point number `f` which can be used in the program.
Note: C is case-sensitive.
A variable which is not supposed to change value can be declared as a constant, e.g.

```
   const float pi = 3.14;
```

This improves efficiency and simplifies debugging.

Arithmetic expressions

A variable is assigned a value using the operator =.
C supports all basic arithmetic operators.

- `a = -b;`
- `a = b + c;`
- `a = b - c;` (equivalent to `a=b+-c;`)
- `a = b * c;`
- `a = b / c;`

Operator precedence as in mathematics. The statement
`a = (1 + 2) * 3 - 4;`
will result in a being assigned the value 5.

Additional mathematical functions in `math.h` (trigonometric, exponential, power, . . .).

Compound assignment

When one is only interested in updating the value of a variable it is possible to use compound assignment.
`a += b;` (`-=` , `*=` , `/=`)
is equivalent to
`a = a + b;` (`-` , `*` , `/`).
`i=i+1` can be conveniently written using the increment operator `++`. `i=i+1` is equivalent to `i++` (postfix) and `++i` (prefix).

Prefix: $i = 1; j = 1; k = j + ++i;$
$i \leftarrow 2$, $k \leftarrow 3$ $(=1+2)$
`i` updated before evaluation

Postfix: $i = 1; j = 1; k = j + i++;$
$i \leftarrow 2$, $k \leftarrow 2$ $(=1+1)$
`i` updated after evaluation

Decrement operator: `--`

Type conversion

All values in C have a specific type.
→ Some mathematical operations can introduce ambiguities.

E.g. what type should the sum of an `int` and a `float` have?

These questions are resolved by type conversion.
Implicit type conversion is done by the compiler when

- A. Type of expressions on left and right hand sides differ (assignment)
- B. Operands in expression of different type

A type cannot be changed during execution.

Case A is handled by converting the value of the right hand side to the type on the left hand side before assignment.

Type conversion (cont.)

Case B is handled by "safe conversion"; if one type can be expressed in the other (e.g. `int` special case of `float`) that type is converted to obtain a more accurate result.

`char < int < long < float < double`

There are some pathological cases. Example:

```
int   i , j;
float f;
i = 3 ; j = 2;
f = i / j;             /* f will hold 1.0 */
f = (float) (i / j);  /* f will hold 1.0 */
f = (float) i / j;    /* f will hold 1.5 */
f = i / (float) j;    /* f will hold 1.5 */
```

Sometimes it is advisable to do an explicit casting,

(type) expression

which converts the value of the expression to the given type.

Formatted output (`stdio.h`)

Output in C is written to output streams.
The two most common are `stdout` (for standard output, usually screen) and `stderr` (for error messages).
Output written to `stderr` is shown as soon as the statement is executed.

A programmer can define new streams.

```
int printf(const char *format, ...);
int fprintf(FILE *stream,
            const char *format, ...);
```

`fprintf` allows the programmer to specify the output stream.
`printf` writes to `stdout`.

The appearance of the output is specified by the format string.
The ellipsis (...) should be replaced by the values to print.

Formatted output (cont.)

A simple example:

`printf("Area = %f\n", pi*r*r);`

`"Area = %f\n"` is a format string instructing the computer to write the string `"Area = "` followed by a floating point number (`pi*r*r` evaluated) and a linebreak to `stdout`.
The format string must contain a conversion specification for each value to print.
Some common conversion specifiers:

Specifier	Displays
%d	Integers
%f	Floating point numbers
%e	– " – , exponential form
%g	Combination of two above
%c	Characters
%s	Strings

Formatted output (cont.)

The basic specifiers can be modified. The format string %p.qX prints a value of type X with precision q and a minimum field width p (for tables).
The meaning of the word precision depends on X. Typically the number of decimals ($X = f$ or $X = e$) or significant digits ($X = g$ or $X = d$).

Escape sequences instructs the computer to print special characters. Some useful examples:

Sequence	**Displays**
`\n`	New line
`\t`	Tab
`\"`	Double quote (")
`\\`	Backslash (\)
`\?`	Question mark (?)

Formatted input

Input in C is read from input streams.
The most common is `stdin` (standard input, usually keyboard).

```
 int scanf(const char *format, ...);
int fscanf(FILE *stream,
              const char *format, ...);
```

The format string is basically the same as for `fprintf`.
The ellipsis should be replaced by addresses to memory locations where the values are to be stored.

Addresses to variables can be obtained by the address operator `&`.
`&x` returns the address of the variable `x` in memory.

Note: To read a double the format string `%lf` must be used, `%f` will not work.

Input/output – an example

```
#include <stdio.h>

main() {
  const float pi = 3.14;
  float radius;

  printf("Enter radius of circle: ");
  scanf("%f", &radius);

  printf("Area = %e, Circumference = %.5f\n\n",
         pi*radius*radius, 2*pi*radius);
}
```

Compilation of the example program in the file *circle.c*

```
c2m2-20>gcc -o circle circle.c
```

Execution of example program:

```
c2m2-20>./circle
Enter radius of circle: 1
Area = 3.140000e+00, Circumference = 6.28000

c2m2-20>./circle
Enter radius of circle: 4.5
Area = 6.358500e+01, Circumference = 28.26000

c2m2-20>./circle
Enter radius of circle: 2.3e1
Area = 1.661060e+03, Circumference = 144.44000
```

If you do not have the module *gcc* you need to add it with the command

```
module add gcc
```

Write the above line in your file *.modules* found in your homedirectory.

Selection statements

Selection statements allows the program to select different execution paths depending on data.
There are two selection statements in C: `if` and `switch`.

The `if` statement chooses execution path by testing a logical expression,

if (expression) statement else statement

Any valid statement is allowed, including a new `if` statement.
Example, computing $abs(x) = |x|$:

```
if (x > 0)
  absx = x;
else
  absx = -x;
```

Logical expressions

Logical expressions are formulated using (a combination of) relational, equality and/or logical operators.

Each logical expression evaluates to either false (0) or true (1, nonzero).

Relational operators	
Symbol	**Meaning**
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Equality operators	
Symbol	**Meaning**
==	equal to
!=	not equal to

Logical expressions (cont.)

Logical operators	
Symbol	**Meaning**
!	logical negation
&&	logical and
||	logical or

Example: $(0 < n \leq 100$ or $n = 150)$
can be written (for integers n)
$(((n > 0)\ \&\&\ (n <= 100)) || (n == 150))$.

Do not confuse == (equality) with = (assignment) since this may lead to the wrong execution path being chosen and relevant data being overwritten.
`(i == j)` tests whether `i` is equal to `j`.
`(i = j)` assigns `i` the value of `j` and is evaluated as true if `j` is nonzero.

The `switch` statement

```
switch (expression)
{
  case {const expr} : {statements}
  ....
  case {const expr} : {statements}
  default : {statements}
}
```

All expressions must return integer (or character) values.
The constant expressions (or case labels) may not contain variables or function calls.

For simple use of `switch` the last statement in each case group should be `break;`.

The switch statement (cont.)

Can use the switch statement to avoid cascaded if statements.

Example: Solve polynomial equation of degree 1 or 2.

Cascaded if statements (top) and switch statement (bottom):

```
if (degree == 1)
  {/* Solve eqn of degree 1*/}
else if (degree == 2)
  {/* Solve eqn of degree 2 */}
else
  {/* Print error message */}
```

```
switch (degree) {
  case 1:  /* Solve degree 1*/; break;
  case 2:  /* Solve degree 2*/; break;
  default: /* Print error message */;
}
```

Iteration statements (loops)

Iteration statements are used to repeatedly execute a statement.
There are three iteration statements in C: `while`, `do` and `for`.

$\boxed{\text{while (expression) statement}}$

`while` executes the statement as long as the expression is true.

Example, computing powers of 2 smaller than 100:

```
  int n   = 0;
  int pow = 1;
  while (pow < 100) {
    printf("%3d   %3d\n", n, pow);
    n = n + 1;
    pow *= 2;
  }
```

do loops

$\boxed{\text{do statement while (expression)}}$

do loops are essentially `while` loops, but the controlling expression is tested after the statement is executed.

Example, do loop written as `while` loop:

```
i=1;
while ( i ) {
   statement
   i = expression;
}
```

for loops

$\boxed{\text{for (expr1 ; expr2 ; expr3)   statement}}$

The syntax of `for` loops is more complicated than that of `while` and do loops, but allows the compiler to produce faster executables.

The expressions should be interpreted as follows:

expr1	Executed before starting iteration.
expr2	Iterate while expression is true.
expr3	Executed at end of each iteration

Example, `for` loop written as `while` loop:

```
expr1;
while ( expr2 ) {
   statement
   expr3;
}
```

for loops (cont.)

Example, computing $n!$

```
fac = 1;
for (i=1 ; i<=n ; i++)
  fac = fac*i;
```

expr1 and expr3 may contain several statements separated by commas.

Example, computing powers of 2 smaller than 100:

```
for (n=0,pow=1 ; pow<100 ; n++,pow*=2)
  printf("%3d   %3d\n", n, pow);
```

Functions in C

Functions in C can be viewed as small programs on their own.

Functions have several advantages:

- Less code duplication – easier to read / update programs
- Simplifies debugging – each function can be verified separately
- Reusable code – the same functions can be used in different programs (e.g. `printf`)

Related functions are often collected in libraries:
e.g. the C standard library, BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage).

Function definition

```
return-type function-name(parameters) {
     declarations
     statements
     return value;
}
```

return-type	type of value returned by function or `void` if none
`function-name`	unique name identifying function
parameters	comma-separated list of types and names of parameters
value	value returned upon termination (`return` value; is not needed if return-type is `void`)

Function definition (cont.)

The list of parameters is a declaration on the form

$\boxed{\text{type\_1 par\_1, \ldots, type\_n par\_n}}$

and represents external values needed by the function.

The list of parameters can be empty.

All declarations and statements that are valid in the main program can be used in the function definition, where they make up the function body.

Example, computing averages

```
double average(double x, double y) {
  return (x+y)/2;
}
```

Function declaration

Like variables a function must be declared before it can be used (called). The declaration of a function resembles the definition,

$\boxed{\text{return-type function-name(parameter types);}}$

The function body is replaced by a semi-colon, and parameters need not be named, it is sufficient to specify their types.

Note: If the function definition precedes the first call the declaration is not needed.

Function calls

$\boxed{\text{variable=function-name(arguments);}}$

variable is assigned the return-value of the function.

The arguments are values with types corresponding to the parameters of the function.
For example,

```
main() {
  double a=1;
  double avg;

  avg = average(a, 3.0);
}
```

If the types of an argument and parameter do not agree, implicit type conversion is used as for assignment.

Function calls (cont.)

When a function is called the value of the argument is copied to the corresponding parameter. Changes made to the parameter inside the function will *not* affect the argument (call by value).
Example, what is the output of the program?

```
#include <stdio.h>

void decrease(int i) {
  i--;
  printf("%d ", i);
}

main() {
  int i=1;
  printf("%d ", i);
  decrease(i);
  printf("%d\n", i);
}
```

Passing arguments by value or reference?

Consider the transformation from polar to Cartesian coordinates, $(r, \varphi) \to (x, y)$. Given two values we wish to compute two new, but functions in C can only return one value.

We can either write separate functions for computing x and y or pass arguments by reference.
If an argument is passed by reference, any changes made to the parameter inside the function will affect the value of the argument outside.

Thus we could pass the function r, φ by value and x, y by reference. Any changes made to x and y inside the function would also apply to the arguments.

We will leave this subject for now and return when we discuss pointers.

Variable scope (example)

```
#include <stdio.h>
int i = 0;              /* Declaration A */
int main() {
  printf("%d ", i);     /* A */
  {
    int i = 1;          /* Declaration B */
    printf("%d ", i);   /* B */
    {
      int i = 2;        /* Declaration C */
      printf("%d ", i); /* C */
      i++;              /* C */
      printf("%d ", i); /* C */
    }
    printf("%d ", i);   /* B */
  }
  printf("%d\n", i);    /* A */
}
```

gives the output 0 1 2 3 1 0

Variable scope

Each identifier is visible only within a certain context, or scope. This is the reason for passing arguments.

There are two kinds of scope in C: local and file scope.

Variables declared inside a block (compound statement) are visible to the end of the block and have local scope. Local scopes can be nested.

Variables declared outside any function definition are visible to the end of the file and have file scope. These global variables should be avoided if possible (simplifies debugging).

Scope rule: When a declaration names a visible identifier, the new declaration "hides" the old and the identifier takes on a different meaning.

An example - Newton's method

Newton's method for solving nonlinear equations $f(x) = 0$,

$$\text{Until convergence: } x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

```
#include <stdio.h>
#include <math.h>

double f(double x) {return (x-cos(x));}
double fprime(double x) {return (1+sin(x));}

main() {
  double dx=1, x=0.5;
  int    i=0;

  while (fabs(dx)>1e-12) {
    dx = -f(x) / fprime(x);
    x  = x + dx;
    printf("%3d %17.12f %15e\n", ++i, x, dx);
  }
}
```

An example - Newton's method

Execution of example program:

```
> gcc newton.c -o newton -lm
> newton
  1     0.755222417106    2.552224e-01
  2     0.739141666150   -1.608075e-02
  3     0.739085133921   -5.653223e-05
  4     0.739085133215   -7.056461e-10
  5     0.739085133215   -3.064197e-17
```

Note: This program would not have converged if `float` had been used instead of `double`. (Tolerance must be $>\approx 5 \cdot 10^{-8}$ for convergence.)