Informatica e Sistemi in Tempo Reale

Introduction to C programming

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

October 25, 2010

Outline

Outline

My first C program

- Let's start with a classic:

hello.c

```
#include <stdio.h>
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

include includes definitions for library functions (in this case, the `printf()` function is defined in header file *stdio.h*)

main function this function must always be present in a C program. It is the first function to be invoked (the *entry point*)

return end of the function, returns a value to the shell

How to compile and run the program

- The C language is a compiled language
  - It means that the above program must be *translated* into a binary code before being executed
- The *compiler* does the job
  - reads the *source* file, translates it into binary code, and produces an *executable* file
  - In Linux, the following command line produces executable file *hello* from source file *hello.c*

```
gcc hello.c -o hello
```

  - In Windows (with DevC++), you must *build* the program
- When you run the program (from a Linux shell, type `./hello`, from Windows, click on Run), you obtain:
  - (in Windows you may not be able to see the output because the shell is automatically closed!)

```
Hello world!
```

Compiling the code

- The translation from high-level language to binary is done by the compiler (and the linker)
  - the **compiler** translates the code you wrote in the source file (*hello.c*)
  - the **linker** links external code from libraries of existing functions (in our case, the `printf()` function for output on screen)

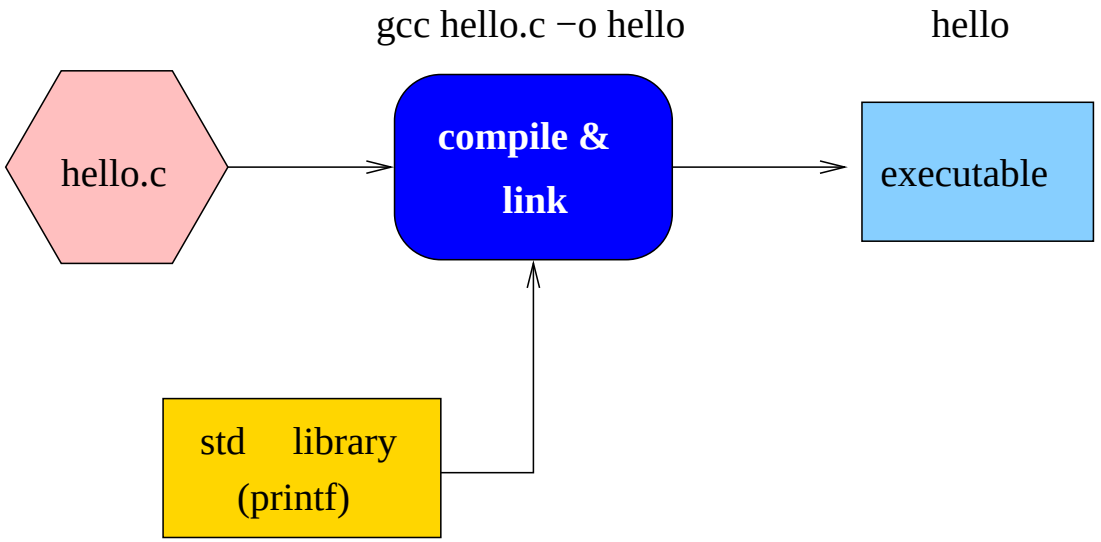

Figure: Compiling a file

Multiple source files

- A program can consist of multiple source files
- Every source file is called *module* and usually consists of a set of well-defined functions that work together
- every source file is compiled separately (it is a *compilation unit*) to produce an *object file* (extension: *.o* or *.obj*)
- all objects files and libraries are then *linked* together to produce an executable
- We will see later how it works

Running a program

- To execute a program, you must tell the Operating System to
  - load the program in main memory (RAM)
  - start executing the program instructions sequentially
- The OS is itself a program!
  - It is a *high-order* program that controls the execution of user programs
- The OS can:
  - Execute several user programs *concurrently* or *in parallel*
  - *suspend* or *kill* a user program
  - coordinate and synchronize user programs
  - let them communicate and exchange data
  - and many other things!

Outline

Declarations, functions, expressions

- A C program is a sequence of global declarations and definitions
  - declarations of *global variables* and *functions*
  - definitions of variables and functions
  - often, declarations are implicit (the definition is an implicit declaration)
  - Examples:

```
int a;              // declaration + definition
int b = 10;         // declaration + definition + init

int f(int);         // declaration only

int f(int p)        // definition
{
    ...
}

int g()             // declaration + definition
{

}
```

Functions

- The code goes inside functions
- There must be always at least one definition of a function called `main`
  - In the *hello* example:

hello.c

```
{
    printf("Hello world!\n");
    return 0;
}
```

Anatomy of the `main` function

- There can be another form of main function:

```
int main(int argc, char *argv[])
{
    ...
}
```

- `main` is the function name, and must be unique in a program
  - there cannot be two functions with the same name
- **`int`** is the return type (will see later)
- between () parenthesis we have the list of parameters with their type, separated by commas:
  - in the example above, two parameters, `argc` and `argv`
- between {} parenthesis, we have the function body:
  - the code that is executed when the function is *called*
- The OS implicitly calls the `main` function when the program is launched
  - the `main` function is also called the program *entry point*

Outline

Variables and types

- A variable is a location in memory with a *symbolic name*
- A variable is used as temporary or permanent storage of data to perform complex computation
- In C, every variable must have a *type*
- Predefined types in C:

int	an integer number (usually 32 bits)
char	a ASCII character (8 bits)
float	floating point number, single precision (32 bits)
double	floating point number, double precision (64 bits)

- A type dictates the variable range (or domain) (from the number of bits) and the operations you can perform on a variable

Variable definition

- Usually, declaration and definition coincide for variables
- The definition consists of the type keyword followed by the name of the variable, followed by the “;” symbol
- Examples

```
int a;              /* an integer variable of name a      */
double b;           /* a double-precision floating point */
char c;             /* a character                        */
...

a = 10;             /* assignment: a now contains 10      */
b = b + 1.5;        /* after assignment, b is equal to
                       the previous value of b plus 1.5  */
c = 'a';            /* c is equal to the ASCII value of
                       character 'a'                      */
```

Constants

- Constants are numeric or alphabetic values that can be used in operations on variables or in functions
- Example:

```
const double pi = 3.1415; /* a double precision constant */
int a = 325;              /* 325 is a constant integer   */
char c = '?';             /* '?' is a constant character */
printf("Hello world!\n"); /* "Hello world!\n" is a       *
                           * constant string             */
```

Variable names

- Variable names cannot start with a number
- cannot contain spaces
- cannot contain special symbols like '+', '-', '*', '/', '%', etc.
- cannot be arbitrarily long (255 char max)
- cannot be equal to reserved keywords (like **`int`**, **`double`**, **`for`**, etc.)

Variable initialization

- It is possible to assign an initial value to a variable during definition
- If you do not specify a value, the initial value of the variable is undefined
- It is good programming practice to always initialize a variable
  - Many programming errors are due to programmers that forget to initialize a variable before using it

```
int a = 0;              /* the initial value is 0  */
int i;                  /* undefined initial value */
int b = 4;

b = i + 5;              /* error! the value of i is not defined! */
```

Operations on variables

- The basic arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
%	modulus (remainder of the integer division)

- Notes:
  - when division is applied to integers, the result is an integer (it truncates the decimal part)
  - modulus can only be applied to integers
  - multiplication, division and modulus have precedence over addition and subtraction
  - to change precedence, you can use parenthesis

Expressions

- A C program is a sequence of expressions, and expression is a combination of operators on variables, constants and functions
- Examples of expressions:

```
/* definitions of variables */
int a, b;
int division;
int remainder;

double area_circle;
double radius;
...

/* expressions */
a = 15;
b = 6;
division = a / b;
remainder = a % b;
radius = 2.4;
area_circle = 3.14 * radius * radius;
```

Assignment and expressions

- Assigning a value to a variable is itself an expression

```
area_circle = 3.14 * radius * radius;
```

- The above expression is composed by three elements:
  - the operator is =
  - the left operand must always be a variable name (cannot be another expression!)
  - the right operand can be any expression, (in our case two multiplications)
  - the right operand is evaluated first, and then the result is assigned to the left operand (the variable)

```
area_circle / 3.14 = radius * radius
```

- the code above is illegal! Why?

Assignment expressions

- The following expression is perfectly legal:

```
int a, b;

b = a = 5;
```

Assignment expressions

- The following expression is perfectly legal:

```
int a, b;

b = a = 5;
```

- You must read it from right to left:
  - a=5 is first evaluated by assigning value 5 to variable a; the result of this expression is 5

Assignment expressions

- The following expression is perfectly legal:

```
int a, b;

b = a = 5;
```

- You must read it from right to left:
  - a=5 is first evaluated by assigning value 5 to variable a; the result of this expression is 5
  - then, the result is assigned to variable b (whose value after assignment is hence 5)

Assignment expressions

- The following expression is perfectly legal:

```
int a, b;

b = a = 5;
```

- You must read it from right to left:
  - a=5 is first evaluated by assigning value 5 to variable a; the result of this expression is 5
  - then, the result is assigned to variable b (whose value after assignment is hence 5)
- What are the values of a and b after the following two expressions?

```
int a, b;

b = (a = 5) + 1;

b = a = 5 + 1;
```

Outline

Formatted output

- To output on screen, you can use the `printf` library function

exprintf.c

```
/* fprintf example */
#include <stdio.h>

int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d %ld\n", 1977, 650000);
   printf ("Preceding with blanks: %10d \n", 1977);
   printf ("Preceding with zeros: %010d \n", 1977);
   printf ("Some different radixes: %d %x %o %#x %#o \n",
           100, 100, 100, 100, 100);
   printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
   printf ("Width trick: %*d \n", 5, 10);
   printf ("%s \n", "A string");
   return 0;
}
```

Formatted Input

- To input variables from the keyboard, you can use the `scanf` library function

exscanf.c

```
/* scanf example */
#include <stdio.h>

int main ()
{
  char str [80];
  int i;

  printf ("Enter your family name: ");
  scanf ("%s",str);
  printf ("Enter your age: ");
  scanf ("%d",&i);
  printf ("Mr. %s , %d years old.\n",str,i);
  printf ("Enter a hexadecimal number: ");
  scanf ("%x",&i);
  printf ("You have entered %#x (%d).\n",i,i);

  return 0;
}
```

Outline

Exercises

1. Write a program that asks the user to enter the radius of a circle, computes the area and the circumference
   - define variables and initialize them
   - use scanf to input radius variable
   - compute the values
   - formatted input on screen
2. Write a program that asks for two integer numbers a and b, computes the quotient and the remainder, and prints them on screen

Outline

Shortcuts

- It is possible to combine assignment with common operators, as follows:

```
a += 5;          // equivalent to a = a + 5;
```

Shortcuts

- It is possible to combine assignment with common operators, as follows:

```
a += 5;          // equivalent to a = a + 5;
```

```
x /= 2;          // equivalent to x = x / 2;
```

Shortcuts

- It is possible to combine assignment with common operators, as follows:

```
a += 5;          // equivalent to a = a + 5;
```

```
x /= 2;          // equivalent to x = x / 2;
```

```
y *= x + a;      // equivalent to y = y * (x+a);
```

Shortcuts

- It is possible to combine assignment with common operators, as follows:

```
a += 5;          // equivalent to a = a + 5;
```

```
x /= 2;          // equivalent to x = x / 2;
```

```
y *= x + a;      // equivalent to y = y * (x+a);
```

- In general

```
var <op>= <expr>;  // equivalent to var = var <op> (<expr>);
```

Increment / decrement

- If you just need to increment/decrement, you can use the following shortcuts

```
x++;            // equivalent to x = x + 1;
++x;            // equivalent to x = x + 1;

y--;            // equivalent to y = y - 1;
--y;            // equivalent to y = y - 1;
```

Increment / decrement

- If you just need to increment/decrement, you can use the following shortcuts

```
x++;            // equivalent to x = x + 1;
++x;            // equivalent to x = x + 1;

y--;            // equivalent to y = y - 1;
--y;            // equivalent to y = y - 1;
```

- Of course, it can only be used on variables;

```
(a+b)++;     // compiler error! cannot increment
             // an expression
x = (a+b)++; // error again: use x = (a+b)+1;
```

Pre and post-increment

- What is the difference between x++ and ++x?

Pre and post-increment

- What is the difference between `x++` and `++x`?
- They are both expressions that can be used inside other expressions (like assignment), as follows;

```
int a, x;
x = 5;

a = ++x; // what is the value of a after the assignment?
```

Pre and post-increment

- What is the difference between `x++` and `++x`?
- They are both expressions that can be used inside other expressions (like assignment), as follows;

```
int a, x;
x = 5;

a = ++x; // what is the value of a after the assignment?
```

- The only difference is the value of the expression:
  - `x++` has the value of `x` **before** the increment;
  - `++x` has the value of `x` **after** the increment;

Pre and post-increment

- What is the difference between x++ and ++x?
- They are both expressions that can be used inside other expressions (like assignment), as follows;

```
int a, x;
x = 5;

a = ++x; // what is the value of a after the assignment?
```

- The only difference is the value of the expression:
  - x++ has the value of x **before** the increment;
  - ++x has the value of x **after** the increment;

```
x = 5;
a = x++;        // value of a is 5, b is 6
```

Pre and post-increment

- What is the difference between x++ and ++x?
- They are both expressions that can be used inside other expressions (like assignment), as follows;

```
int a, x;
x = 5;

a = ++x; // what is the value of a after the assignment?
```

- The only difference is the value of the expression:
  - x++ has the value of x **before** the increment;
  - ++x has the value of x **after** the increment;

```
x = 5;
a = x++;        // value of a is 5, b is 6
```

```
x = 5;
a = ++x;        // value of a is 6, b is 6
```

Boolean operators

- In there is no *boolean* type
- Every expression with a value equal to 0 is interpreted as *false*
- Every expression with a value different from 0 is interpreted as *true*
- It is possible to use the following boolean operators:

  && logical and operator
  || logical or operator
  ! logical not operator

- It is possible to interpret integer values as booleans and vice versa

```
int a, b, c;
a = 0; b = 5;

c = a && b;   // after assignment, c is 0;
c = a || b;   // after assignment, c is 1;
c = !b;       // after assignment, c is 0;
```

Comparison operators

- These operators compare numbers, giving 0 or 1 (hence a boolean value) as result

`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	equal
`!=`	not equal

Comparison operators

- These operators compare numbers, giving 0 or 1 (hence a boolean value) as result
  - $<$ less than
  - $<=$ less than or equal to
  - $>$ greater than
  - $>=$ greater than or equal to
  - $==$ equal
  - != not equal

```
int a = 7; int b = 10; int c = 7;

int res;

res = a < b;    // res is 1
res = a <= c;   // res is 1
res = a < c;    // res is 0

res = b == c;   // res is 0
```

- (will come back to these later)

Binary operators

- It is possible to do binary operations on integer variables using the following operators:
  - & binary (bit-to-bit) and
  - | binary (bit-to-bit) or
  - ~ binary (bit-to-bit) not (complement)

```
unsigned char a = 1;   // in binary: 0000 0001
unsigned char b = 2;   // in binary: 0000 0010
unsigned char c = 5;   // in binary: 0000 0101
unsigned char d;

d = a & b;             // d is now 0000 0000
d = a & c;             // d is now 0000 0001
d = a | b;             // d is now 0000 0011
d = ~a;                // d is now 1111 1110
```

Outline

Execution flow

- Usually, instructions are executed sequentially, one after the other, until the end of the function

Execution flow

- Usually, instructions are executed sequentially, one after the other, until the end of the function
- However, in many cases we must execute alternative instructions, depending on the value of certain expressions

Execution flow

- Usually, instructions are executed sequentially, one after the other, until the end of the function
- However, in many cases we must execute alternative instructions, depending on the value of certain expressions
- Also, sometimes we need to repeat instructions a number of times, or until a certain condition is verified

Execution flow

- Usually, instructions are executed sequentially, one after the other, until the end of the function
- However, in many cases we must execute alternative instructions, depending on the value of certain expressions
- Also, sometimes we need to repeat instructions a number of times, or until a certain condition is verified
- we need to **control the execution flow**

Outline

If statement

- To select alternative paths, we can use the *if then else* statement
- The general form is the following:

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

- `<expression>` must be a boolean expression;

If statement

- To select alternative paths, we can use the *if then else* statement
- The general form is the following:

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

- `<expression>` must be a boolean expression;
- The statement can be a single code instruction, or a *block* of code:

```
if (<expression>) {
    statement1;
    statement2;
    statement3;
}
```

- A block is a set of statements encloses by curly braces {}

Examples

- here are two example of usage of *if*

```
int x;
...
if (x % 2)
    printf("number %d is odd\n", x);
```

```
double a;

if (a < 0) {
    printf("a is negative!\n");
    a = -a;
    printf("a is now positive\n");
}
```

Complete form

- In its most complete form:

```
if (<expression>)
    statement1;
else
    statement2;
```

- Of course, both `statement1` and `statement2` can be blocks of statements;

```
if (x > 0) {
    if (y > 0)
        printf("Northeast.\n");
    else
        printf("Southeast.\n");
}
else {
    if (y > 0)
        printf("Northwest.\n");
    else
        printf("Southwest.\n");
}
```

Statements

- A statement can be:
  - an expression;
  - a *if then else* construct;
  - a block of statements (recursive definition!)
- Expressions and statements are not the same thing!
  - You can use expressions wherever you can use a statement
  - You cannot use a statement where you see "expression"!
- For example, you cannot use a statement inside a if condition!
- But you can use another if as a statement

Statements - 2

- You can write the following:

```
if (x > 0) if (y > 0) printf("north east\n");
    else printf("south east\n");
else if (y > 0) printf("north west\n");
         else printf("south west\n");
```

- here *if* is used as a statement inside another if

Statements - 2

- You can write the following:

```
if (x > 0) if (y > 0) printf("north east\n");
    else printf("south east\n");
else if (y > 0) printf("north west\n");
         else printf("south west\n");
```

- here *if* is used as a statement inside another if
- You cannot write the following:

```
if (if (x > 0)) ...
```

- in facts, an if condition can only be an expression!

Statements - 2

- You can write the following:

```
if (x > 0) if (y > 0) printf("north east\n");
    else printf("south east\n");
else if (y > 0) printf("north west\n");
         else printf("south west\n");
```

- here *if* is used as a statement inside another if
- You cannot write the following:

```
if (if (x > 0)) ...
```

- in facts, an if condition can only be an expression!
- Remember:
  - An expression has always a (numerical) value which is the result of an operation

Statements - 2

- You can write the following:

```
if (x > 0) if (y > 0) printf("north east\n");
    else printf("south east\n");
else if (y > 0) printf("north west\n");
         else printf("south west\n");
```

- here *if* is used as a statement inside another if
- You cannot write the following:

```
if (if (x > 0)) ...
```

- in facts, an if condition can only be an expression!
- Remember:
  - An expression has always a (numerical) value which is the result of an operation
  - 0 is interpreted as false, any other number is interpreted as true

Statements - 2

- You can write the following:

```
if (x > 0) if (y > 0) printf("north east\n");
    else printf("south east\n");
else if (y > 0) printf("north west\n");
         else printf("south west\n");
```

- here *if* is used as a statement inside another if
- You cannot write the following:

```
if (if (x > 0)) ...
```

- in facts, an if condition can only be an expression!
- Remember:
  - An expression has always a (numerical) value which is the result of an operation
  - 0 is interpreted as false, any other number is interpreted as true
  - A statement may be an expression (in which case it has a numerical value), or something else

More on if conditions

- To check if variable `i` is between `1` and `10`:

```
if (i <= 10 && i>= 1) ...
```

- or alternatively:

```
if (1 <= i && i <= 10) ...
```

More on if conditions

- To check if variable `i` is between `1` and `10`:

```
if (i <= 10 && i>= 1) ...
```

- or alternatively:

```
if (1 <= i && i <= 10) ...
```

- Don't use the following:

```
if (1 <= i <= 10) ...
```

- (what happens? check out `./examples/01.intro_c-examples/condition1.c`)

Common mistakes

- One common mistake is the following:

```
int a = 5;
if (a = 0) printf("a is 0\n");
else printf("a is different from 0\n");
```

- What does the code above print on screen? (see `./examples/01.intro_c-examples/condition2.c`)

Common mistakes

- One common mistake is the following:

```
int a = 5;
if (a = 0) printf("a is 0\n");
else printf("a is different from 0\n");
```

- What does the code above print on screen? (see `./examples/01.intro_c-examples/condition2.c`)
- The value of expression `a = 0` (which is an assignment, not a comparison!) is 0, i.e. the value of `a` after the assignment
- Probably, the programmer wanted to say something else:

```
if (a == 0) printf("a is 0\n");
else printf("a is different from 0\n");
```

Outline

Loops

- In many cases, we need to execute the same code many times, each time on a different set of values
- Example:
  - Given an integer number stored in variable a, print “number is prime” if the number is prime (divisible only by 1 and by itself)

Loops

- In many cases, we need to execute the same code many times, each time on a different set of values
- Example:
  - Given an integer number stored in variable a, print “number is prime” if the number is prime (divisible only by 1 and by itself)
  - To solve the problem, we need to check the remainder of the division between a and all numbers less than a. If it is always different from 0, then the number is prime

Loops

- In many cases, we need to execute the same code many times, each time on a different set of values
- Example:
  - Given an integer number stored in variable a, print “number is prime” if the number is prime (divisible only by 1 and by itself)
  - To solve the problem, we need to check the remainder of the division between a and all numbers less than a. If it is always different from 0, then the number is prime
  - However, we do not know the value of a before program execution; how many division should we do?

Loops

- In many cases, we need to execute the same code many times, each time on a different set of values
- Example:
  - Given an integer number stored in variable a, print “number is prime” if the number is prime (divisible only by 1 and by itself)
  - To solve the problem, we need to check the remainder of the division between a and all numbers less than a. If it is always different from 0, then the number is prime
  - However, we do not know the value of a before program execution; how many division should we do?

Loops

- In many cases, we need to execute the same code many times, each time on a different set of values
- Example:
  - Given an integer number stored in variable a, print “number is prime” if the number is prime (divisible only by 1 and by itself)
  - To solve the problem, we need to check the remainder of the division between a and all numbers less than a. If it is always different from 0, then the number is prime
  - However, we do not know the value of a before program execution; how many division should we do?
- Solution: use the *while* construct

While loop

- The general form:

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

  - As usual, statement can also be a block of statements

While loop

- The general form:

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

  - As usual, statement can also be a block of statements

- Similar to an if, but the statement is performed iteratively while the condition is “true” (i.e. different from 0)
- Example: sum the first 10 numbers:

```
int sum = 0;
int i = 0;

while (i < 10) {
    sum = sum + i;
    i = i + 1;
}

printf("The sum of the first 10 numbers: %d\n", sum);
```

Break and continue statements

- Sometimes we need to go out of the loop immediately, without completing the rest of the statements. To do this we can use the break statement

```
int i = 0;
while (i < 10) {
    i++;
    if ((i % 5) == 0) break;
    printf("%d is not divisible by 5\n", i);
}
printf("Out of the loop");
```

Break and continue statements

- Sometimes we need to go out of the loop immediately, without completing the rest of the statements. To do this we can use the break statement

```
int i = 0;
while (i < 10) {
    i++;
    if ((i % 5) == 0) break;
    printf("%d is not divisible by 5\n", i);
}
printf("Out of the loop");
```

- Another possibility is to continue with the next iteration without complete the rest of the statements:

```
int i = 0;
while (i < 10) {
    i++;
    if (i % 5 != 0) continue;
    printf("%d is divisible by 5\n", i);
}
printf("Out of the loop\n");
```

Prime numbers

isprime.c

```
int main()
{
    int k, i, flag;

    printf("This program tests if a number is prime\n");
    printf("Insert a number: ");
    scanf("%d", &k);

    flag = 1;
    i = 2;

    while (i < k) {
        if (k % i == 0) {
            printf("%d is a divisor: %d = %d x %d\n", i, k, i, k/i);
            flag = 0;
            break;
        }
        i++;
    }
    printf("%d is ", k);
    if (!flag) printf("not ");
    printf("prime\n");
```

Outline

Loops

- *if then else* and *while* constructs are all we need to program

Loops

- *if then else* and *while* constructs are all we need to program
  - It can be proved in theoretical computer science that with one loop construct and one selection construct, the language is equivalent to a Turing Machine, the simplest and more general kind of calculator

Loops

- *if then else* and *while* constructs are all we need to program
  - It can be proved in theoretical computer science that with one loop construct and one selection construct, the language is equivalent to a Turing Machine, the simplest and more general kind of calculator
- However, sometimes using only *while* loops can be annoying

Loops

- *if then else* and *while* constructs are all we need to program
  - It can be proved in theoretical computer science that with one loop construct and one selection construct, the language is equivalent to a Turing Machine, the simplest and more general kind of calculator
- However, sometimes using only *while* loops can be annoying
- The C language provides two more loop constructs: *for* loops and *do-while* loops

For loop

- The most general form is the following:

```
for(<expr1>; <expr2>; <expr3>) statement;
```

For loop

- The most general form is the following:

```
for(<expr1>; <expr2>; <expr3>) statement;
```

  - `expr1` is also called *initialization*; it is executed before entering the first loop iteration
  - `expr2` is also called *condition*; it is checked before every iteration;
    - if it is false, the loop is terminated;
    - if it is true, the iteration is performed
  - `expr3` is also called *instruction*; it is performed at the end of every iteration

- The most common usage is the following:

```
for (i=0; i<10; i++)
    printf("The value of i is now %d\n", i);
```

Sum the first 10 numbers

```
int n = 10;
int i;
int sum = 0;

for (i=0; i<n; i++) sum += i;

printf("The sum of the first %d numbers is %d\n", n, sum);
```

Prime numbers

isprime2.c

```
int main()
{
    int k, i, flag;

    printf("This program tests if a number is prime\n");
    printf("Insert a number: ");
    scanf("%d", &k);

    flag = 1;

    for (i=2; i<k/2; i++)
        if (k % i == 0) {
            printf("%d is a divisor: %d = %d x %d\n", i, k, i, k/i);
            flag = 0;
            break;
        }

    printf("%d is ", k);
    if (!flag) printf("not ");
    printf("prime\n");
```

Equivalence between for and while

- We can always rewrite any while loop as a for loop, and vice versa

Equivalence between for and while

- We can always rewrite any while loop as a for loop, and vice versa

```
for (expr1; expr2; expr3) statement;
```

can be rewritten as:

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

Equivalence between for and while

- We can always rewrite any while loop as a for loop, and vice versa

```
for (expr1; expr2; expr3) statement;
```

can be rewritten as:

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

- On the other hand, the following while loop;

```
while (expr) statement;
```

can be rewritten as:

```
for( ; expr ; ) statement;
```

Outline

Exercises

1. Given the following for loop, rewrite it as a while loop;

```
int k, i=0; j=8;
for (k=0; k<j; k++) {
    i = k+j;
    j--;
    printf("i is now %d\n", i);
}
```

Exercises

1. Given the following for loop, rewrite it as a while loop;

```
int k, i=0; j=8;
for (k=0; k<j; k++) {
    i = k+j;
    j--;
    printf("i is now %d\n", i);
}
```

2. Write a program that, given an integer number in input, prints on screen all prime factors of the number,

Exercises

1. Given the following for loop, rewrite it as a while loop;

```
int k, i=0; j=8;
for (k=0; k<j; k++) {
    i = k+j;
    j--;
    printf("i is now %d\n", i);
}
```

2. Write a program that, given an integer number in input, prints on screen all prime factors of the number,
   - For example, given 6, prints 2, 3

Exercises

1. Given the following for loop, rewrite it as a while loop;

```
int k, i=0; j=8;
for (k=0; k<j; k++) {
    i = k+j;
    j--;
    printf("i is now %d\n", i);
}
```

2. Write a program that, given an integer number in input, prints on screen all prime factors of the number,
   - For example, given 6, prints 2, 3
   - given 24, prints 2, 2, 2, 3

Exercises

1. Given the following for loop, rewrite it as a while loop;

```
int k, i=0; j=8;
for (k=0; k<j; k++) {
    i = k+j;
    j--;
    printf("i is now %d\n", i);
}
```

2. Write a program that, given an integer number in input, prints on screen all prime factors of the number,
   - For example, given 6, prints 2, 3
   - given 24, prints 2, 2, 2, 3
   - given 150, prints 2, 3, 5, 5

Exercises

1. Given the following for loop, rewrite it as a while loop;

```
int k, i=0; j=8;
for (k=0; k<j; k++) {
    i = k+j;
    j--;
    printf("i is now %d\n", i);
}
```

2. Write a program that, given an integer number in input, prints on screen all prime factors of the number,
   - For example, given 6, prints 2, 3
   - given 24, prints 2, 2, 2, 3
   - given 150, prints 2, 3, 5, 5
   - etc.

Exercises

1. Given the following for loop, rewrite it as a while loop;

```
int k, i=0; j=8;
for (k=0; k<j; k++) {
    i = k+j;
    j--;
    printf("i is now %d\n", i);
}
```

2. Write a program that, given an integer number in input, prints on screen all prime factors of the number,
   - For example, given 6, prints 2, 3
   - given 24, prints 2, 2, 2, 3
   - given 150, prints 2, 3, 5, 5
   - etc.
   - **Suggestion:** use a while loop initially

Exercises: strange for loops

Since an expression can be pretty much everything, you can write lot of strange things with for loops

1. Incrementing 2 variables with the comma operator:

```
int i, j;
for (i=0, j=0;  i < 5; i++, j+=2)
    printf(" i = %d,  j = %d\n", i, j);
```

   - What does the code above print on screen?

2. What the code below prints on screen?

```
int i;
int g=0;
for (i=0; i<10; g += i++);
printf("%d", g);
```