



Scuola Superiore Sant'Anna



A short introduction to the C programming language

Giuseppe Lipari



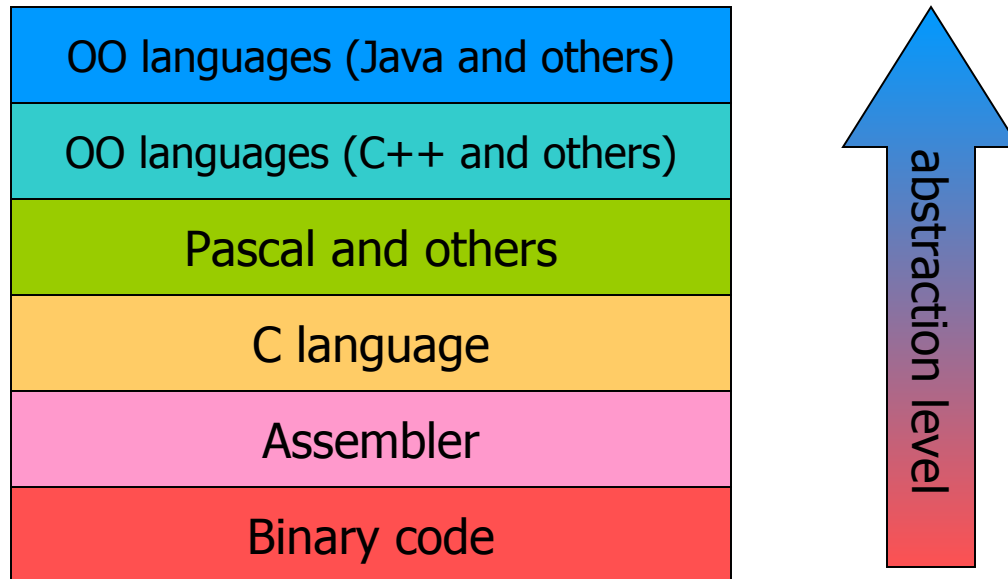
Programming

- programming a computer simply means telling it what to do
- you have to talk in a language that the computer can understand
- The computer accepts list of instructions codified in binary words
- Since it is too difficult to program using bits and bytes directly, we will use an intermediate high level language



High level languages

- C is probably the lowest high-level language
- If we have to draw a hierarchy of languages, we have:





My first C program

- Let's start from a very simple program
- it prints the string "Hello world" on the screen

Includes definitions for the external library functions used in the program (in this case: printf)

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

```
int main()  
{
```

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

```
}
```

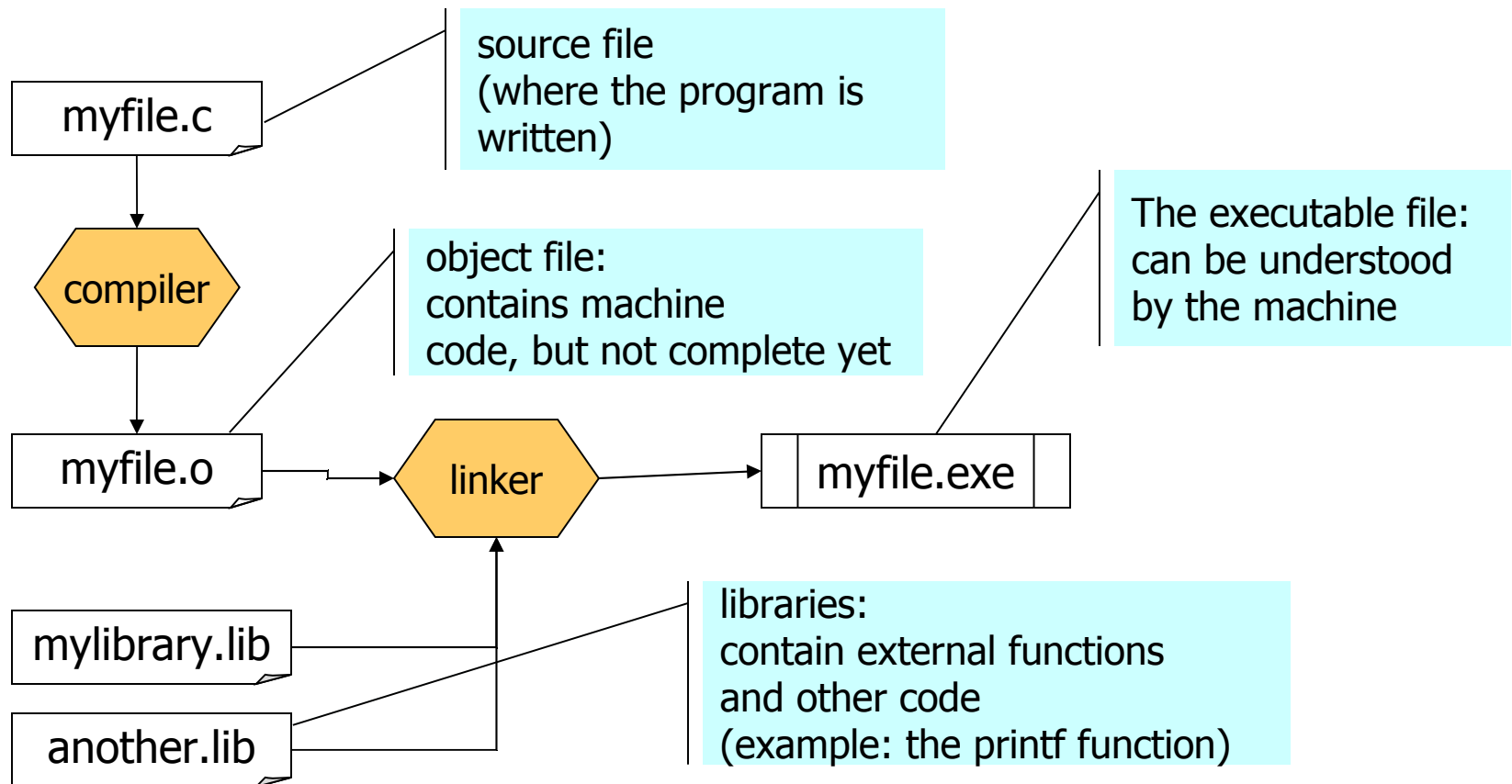
main function: must always be present in a C program

Finish the main function: the program terminates



Compilation, linking, running

- The **compiler** and the **linker** transform the high level code into machine executable code.



Second program

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

```
/* print a few numbers, to illustrate a simple loop */
```

```
int main()
```

```
{
```

```
    int i;
```

```
    for(i = 0; i < 10; i = i + 1)
```

```
        printf("i is %d\n", i);
```

```
    return 0;
```

```
}
```

Comment:
will be ignored by
the compiler

declaration of
an integer variable

a *for* loop:
the following instruction
will be executed 10 times

prints the value of
i on the screen



A few anticipations

- Variables:
 - denote memory locations
 - can assume values
 - can be used to perform complex operations, store results, etc.
- Functions
 - like mathematical functions, they manipulate arguments and variables to produce a result



Variables and Types



Types

- Every variable must have a *type*
- (Some) predefined types in C:
 - `char`: a character
 - `int`: an integer, in the range -32,767 to 32,767
 - `long int`: a larger integer (up to +-2,147,483,647)
 - `short int`: a small integer (in the range -128 to 127)
 - `float`: a floating-point number
 - `double`: a floating-point number, with more precision and perhaps greater range than float
- A type dictates:
 - what kind of operation you can do on the variable
 - what is the range of values the variable can get



Constants

- A constant is just a value that you can use in the program
- Examples:

<code>int a;</code>	<code>/* an integer variable</code>	<code>*/</code>
<code>double b;</code>	<code>/* a floating point variable</code>	<code>*/</code>
<code>char c;</code>	<code>/* a character variable</code>	<code>*/</code>
<code>a = 10;</code>	<code>/* 10 is an integer constant</code>	<code>*/</code>
<code>b = b + 1.5;</code>	<code>/* 1.5 is a floating point constant</code>	<code>*/</code>
<code>c = 'a';</code>	<code>/* 'a' is a character constant</code>	<code>*/</code>
<code>printf("Hello world\n");</code>	<code>/* "Hello world\n" is a string constant</code>	<code>*/</code>



Variable declaration

- Variables must be declared before they can be used
- A declaration tells the compiler the name and type of a variable you'll be using in your program

```
char c;  
int i;  
float f;  
int i1, i2;
```

- It is possible to *initialize* the variables when they are declared
- It is a good programming practice to always initialize variables

```
int i = 1;  
int i1 = 10, i2 = 20;
```



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.)



Variables and memory

- Variables are *allocated* in memory
 - when declaring an int variable, the compiler reserves 4 bytes in memory for the variable
 - that memory will represent the “physical storage” for the variable
 - Every time we operate in the variable, the corresponding memory is used
 - The amount of memory that is allocated depends on the type
 - The interpretation of the memory content depends on the type!



Expressions

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

```
x = 1;  
a && b;  
i = i + 1;  
f(a,b);  
  
c = (a+b)++;  
  
x = y = 2*sqrt(z^3);
```



Arithmetic operators

- The following operators can be used for the basic arithmetic operations
 - + addition
 - - subtraction
 - * multiplication
 - / division
 - % modulus (remainder)
- Notes
 - when division is applied to integers, 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



Assignment operator

- To assign a value or the result of an expression to a variable, use the operator =
- it must have a variable to the left of the =, and a value or an expression to the right

```
x = 1;  
a = b;  
i = i + 1;
```

- The assignment is an operation, and has a “result”, which is the value of the left side variable after the assignment
- You can use the assignment wherever you can use an expression

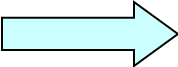
```
c = (a = 5);  
b = (c = c + 1);
```

- what is the value of b after the above expressions?

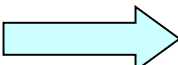


Advanced operators

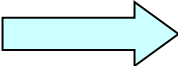
- You can shortcut some operation on a variable like this:

<pre>a = a + 1; x = x / 2; y = y * 4;</pre>		<pre>a += 1; x /= 2; y *= 4;</pre>
---	--	--

- In general

<pre>x = x op (expr);</pre>		<pre>x op= (expr);</pre>
-----------------------------	--	--------------------------

- if you just need to increment (decrement)

<pre>a = a + 1; x = x - 1;</pre>		<pre>a++; x--;</pre>
--------------------------------------	--	--------------------------



Pre and post increment (decrement)

- Two ways of incrementing a variable:
 - pre-increment: `++a`
 - post-increment: `a++`
- The final result is the same
- However, there is a slight difference

```
a = 5;
```

```
b = a++;
```

after the assignment:
a is 6
b is 5

```
a = 5;
```

```
b = ++a;
```

after the assignment:
a is 6
b is 6



Boolean operators

- In C, every expression with a value equal to 0 is interpreted as false
- every expression with a value different from 0 is interpreted as true
- The following operators can be used for the basic logical boolean operations
 - `&&` logical and
 - `||` logical or
 - `!` logical not
- It is possible to interpret integer values as boolean and viceversa

```
int a, b, c;  
a = 1; b = 0;  
  
c = a && b;
```

after the assignment:
c is 0



Statements and control flow



The if statement

- The general form is the following

```
if ( expression )  
    statement
```

- The statement can be a single expression, or a *block* of statements:

```
if( expression ) {  
    statement1  
    statement2  
    statement3  
}
```

- A block is a set of statements enclosed in a pair of curly braces { }



Examples

```
int n;  
...  
if ( (n % 2) == 0 )  
    printf ("number %d is even \n", n);
```

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



else

- The most complete form includes a else statement:

```
if( expression )  
    statement1  
else  
    statement2
```

- Example:

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



logic operators

- To compare numbers:
 - < less than
 - <= less than or equal
 - > greater than
 - >= greater than or equal
 - == equal
 - != not equal
- Logic operators
 - && and
 - || or
 - ! not (takes one operand; ``unary"')



Example

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

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

- The following is wrong:

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

- (The compiler will complain...)

```
if (a = 0) ...
```



Results of boolean expression

- Every expression that evaluates to 0 is “false”
- Every expression that evaluates to something different than 0 is “true”
- Example:

```
int n;  
int sum, average;  
...  
if (n) average = sum / n;
```

- `n` is an expression with value equal to the value of variable `n`
- if `n` is zero, the expression evaluates to “false”, and the division is not performed; the division is done only if `n` is not 0
- It can also be written as follows

```
if (n != 0)  
    average = sum / n;
```



A typical error

- A typical error: to write `=` instead of `==` in a boolean expression

```
if (a = 0)  
    statement;
```

- The statement will never be executed! why?



while

```
while ( expression ) statement
```

- 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);
```



The for loop

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

- Example:

```
for (i=0; i<10; i=i+1) instr
```

Initialization:

variable *i* is set equal to 0
this instruction is executed
only one time, before the
first iteration starts

Condition:

the expression is evaluated
before the start of every iteration.
If the condition is *true* then the
iteration is performed,
otherwise the loop is finished

Instruction:

after each iteration, the instruction
is executed. In this case, variable
i is incremented by 1.



Equivalence between while and for

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

- can be rewritten as:

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

- similarly:

```
while ( expression ) statement
```

- can be rewritten as:

```
for ( ; expression ; ) statement
```



Jumping out of a loop

- **break**: jumps out of the inner loop
 - (skipping the remaining statements)
- **continue**: jumps at the next loop iteration
 - (skipping the remaining statements)



Example: what does this program?

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

```
int main()
```

```
{
```

```
    int i, j;
```

```
    printf("%d\n", 2);
```

```
    for (i = 3; i <= 100; i = i + 1) {
```

```
        for (j = 2; j < i; j = j + 1) {
```

```
            if (i % j == 0)
```

```
                break;
```

```
            if (j > sqrt(i)) {
```

```
                printf("%d\n", i);
```

```
                break;
```

```
            }
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```

Outer loop on variable i

Inner loop on variable j
j ranges from 2 to i

If j divides i, then i is not prime:
exits from inner loop

If j is greater than sqrt(i),
it cannot divide i anymore:
i is prime, and we can
exit from inner loop



switch / case

- When you have multiple choices, depending on the value of a variable, you can use a switch/case rather than a sequence of if – else

```
/* Estimate a number as none, one, two, several, many */
int estimate(number)
{
    switch(number) {
        case 0 :
            printf("None\n");
            break;

        case 1 :
            printf("One\n");
            break;

        case 2 :
            printf("Two\n");
            break;

        case 3 :
        case 4 :
        case 5 :
            printf("Several\n");
            break;

        default :
            printf("Many\n");
            break;
    }
}
```



Basic Input/Output



Basic input/output

- Output of characters

```
int putchar(int c);
```

output of 1
character on screen

```
int puts(const char *s);
```

output of the string
on screen (until the
terminating 0)

- Input of characters

```
int getchar(void);
```

```
char* gets(char *s);
```

very dangerous! Do
not use it

```
char *fgets(char *s, int n, FILE *stream)
```

Prefer this one!



printing on screen

- `printf` stands for **print formatted**
- It is a special kind of function
 - it takes a variable number of arguments
 - the first argument is always a string
 - the other ones depend on what the string contains
- The string may contain one of the following special “format specifiers”:
 - `%d` print an int argument in decimal
 - `%ld` print a long int argument in decimal
 - `%c` print a character
 - `%s` print a string
 - `%f` print a float or double argument
 - `%e` same as `%f`, but use exponential notation
 - `%g` use `%e` or `%f`, whichever is better
 - `%o` print an int argument in octal (base 8)
 - `%x` print an int argument in hexadecimal (base 16)
 - `%%` print a single `%`



Examples

```
printf("%c %d %f %e %s %d%%\n", '1', 2, 3.14, 56000000., "eight", 9);
```

- This will print on screen the following

```
1 2 3.140000 5.600000e+07 eight 9%
```

```
printf("Hello, ");  
printf("world!\n");
```



```
Hello world!
```



Input from keyboard

- `scanf` stands for scan **formatted**
- like `printf`, it is a special kind of function
 - it takes a variable number of arguments
 - the first argument is always a string
 - the other ones depend on what the string contains
- the string format specifiers are similar to `printf`



Example with scanf

```
/* 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;
}
```

Pay attention to
the '&' symbol!

Pay attention to
the '&' symbol!



Arrays



Declaration

- Instead of single variables, we can declare *arrays*
- These are all variables of the same type, with the same name, which can be addressed through an index

```
int i;  
int a[10];  
...  
a[0] = 10;  
a[1] = 20;  
a[2] = a[0] + a[1];
```

- If an array has **N** elements, index starts at **0** and ends at **N-1**
- in the above example, the last valid element is **a[9]**
- the index can be a variable:

```
int i;  
  
for(i = 0; i < 10; i = i + 1) a[i] = 0;
```



Example

- Counts the frequency of occurrence of a number when rolling two dices

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

int main()
{
    int i;
    int d1, d2;
    int a[13]; /* uses [2..12] */

    for(i = 2; i <= 12; i = i + 1) a[i] = 0;

    for(i = 0; i < 100; i = i + 1) {
        d1 = rand() % 6 + 1;
        d2 = rand() % 6 + 1;
        a[d1 + d2] = a[d1 + d2] + 1;
    }

    for(i = 2; i <= 12; i = i + 1)
        printf("%d: %d\n", i, a[i]);

    return 0;
}
```



Strings

- A string is an array of characters

```
char s[10];  
  
char s1[] = "Hello world";  
char myname[] = "Giuseppe Lipari";
```

- The latest element of a string must be character 0
- 0 is the “terminator” of a string
- if you want to store a string of 10 characters, reserve 11 elements in the array!
 - in the example, `s1` has 12 elements
 - `myname` has 16 elements



manipulating strings

- **To copy strings:**
 - **int strcpy(char *s1, char *s2)**
 - copies contents of s2 into s1
- **To compare strings**
 - **int strcmp(char *s1, char *s2)**
 - alphabetical comparison ($a < b$ and $a < A$)
 - returns negative if $s1 < s2$
 - returns 0 is equal
 - returns positive if $s1 > s2$
- **To append strings**
 - **int strcat(char *s1, char *s2)**
 - appends s2 to s1
- **To compute the length**
 - **int strlen(char s1)**
 - counts the number of characters before the string terminator (0)
- **To print strings on screen**
 - **printf("%s", s1);**
 - prints s1 on the screen



Functions

Definition

- A function is defined by
 - a name
 - a list of arguments (or parameters)
 - a body (list of statements)
 - a return value

function
name

type of the
return value

function
body

returned
value

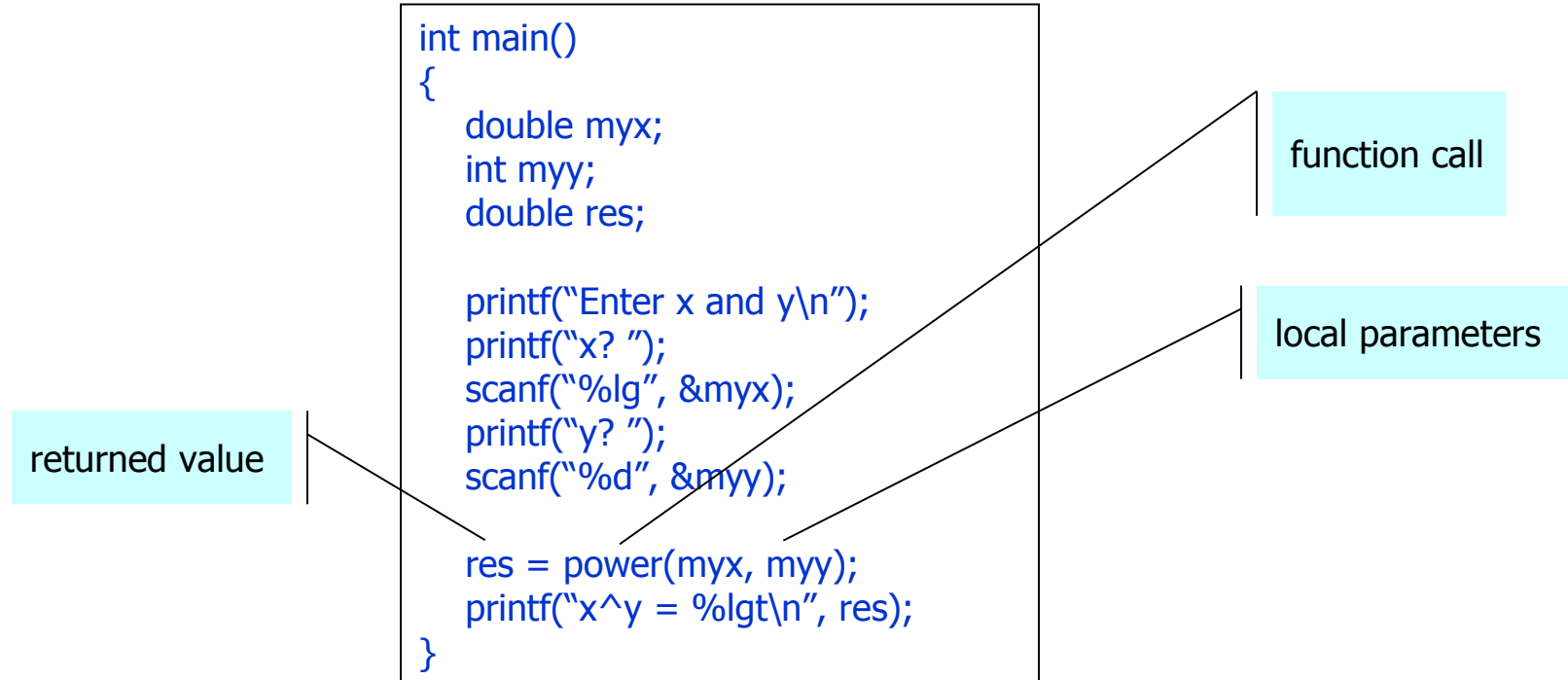
```
/* returns the power of x to y */  
double power(double x, int y)  
{  
    int i;  
    double result = 1;  
  
    for (i=0; i < y; i++)  
        result = result * x;  
  
    return result;  
}
```

second
parameter

first paramter



function call



- Notice that `printf` and `scanf` are also functions (external)
- `main` is also a function
 - automatically called at the program startup



local parameters

- The local parameters cannot be changed by the function

```
int multbytwo(int x)
{
    x = x * 2;
    return x;
}

int main()
{
    ...
    i = 5;
    res = multbytwo(i);
    /* how much is i here? */
    ...
}
```

- x is just a copy of i
- if you change x, you change the copy, not the original value
- There is one exception to this rule:
- arrays!



passing arrays as parameters

```
void swap(int a[])
{
    int tmp;
    tmp = a[0];
    a[0] = a[1];
    a[1] = tmp;
    return;
}

int main()
{
    int my[2] = {1,5}
    printf ("before swap: %d %d",
           my[0], my[1]);

    swap(my);

    printf ("after swap: %d %d",
           my[0], my[1]);
}
```

- The array is not copied
- modification on array *a* are reflected in modification on array *my*
 - (this can be understood better when we study pointers)
- Notice also:
 - the *swap* function does not need to return anything: so the return type is *void*
 - the array *my* is initialized when it is declared



function prototypes

- Functions (and any other object like variables) must be declared in the file before being used
- Sometimes, we may need to use a function before it is actually completely specified
- We can use function prototypes
 - it is a function declaration without its definition (body)
 - in the prototype you need to specify the type of each parameter, but you can avoid specifying the parameter's names

```
int multbytwo(int);  
void swap(int a[]);  
double power(double, int)
```



modules

- Writing every function that is needed to a program into a single file, is not a good idea
- To structure the program, it is much better to divide the code into different files
- Two types of files
 - Source files (*.c) they usually contain code
 - Header files (*.h) they usually contain declarations



From sources to executable: update

- Suppose your program is divided into 3 source files and 2 header files

swap.h

```
/* prototype of swap  
   function */  
void swap(int a[]);
```

power.h

```
/* prototype of power  
   function */  
double power(double, int);
```

prog.c

```
#include "swap.h"  
#include "power.h"  
  
int main()  
{  
    ...  
    power(5.0, 2);  
    ...  
    swap(myarray);  
    ...  
}
```

swap.c

```
#include "swap.h"  
  
void swap(int a[])  
{  
    ...  
}
```

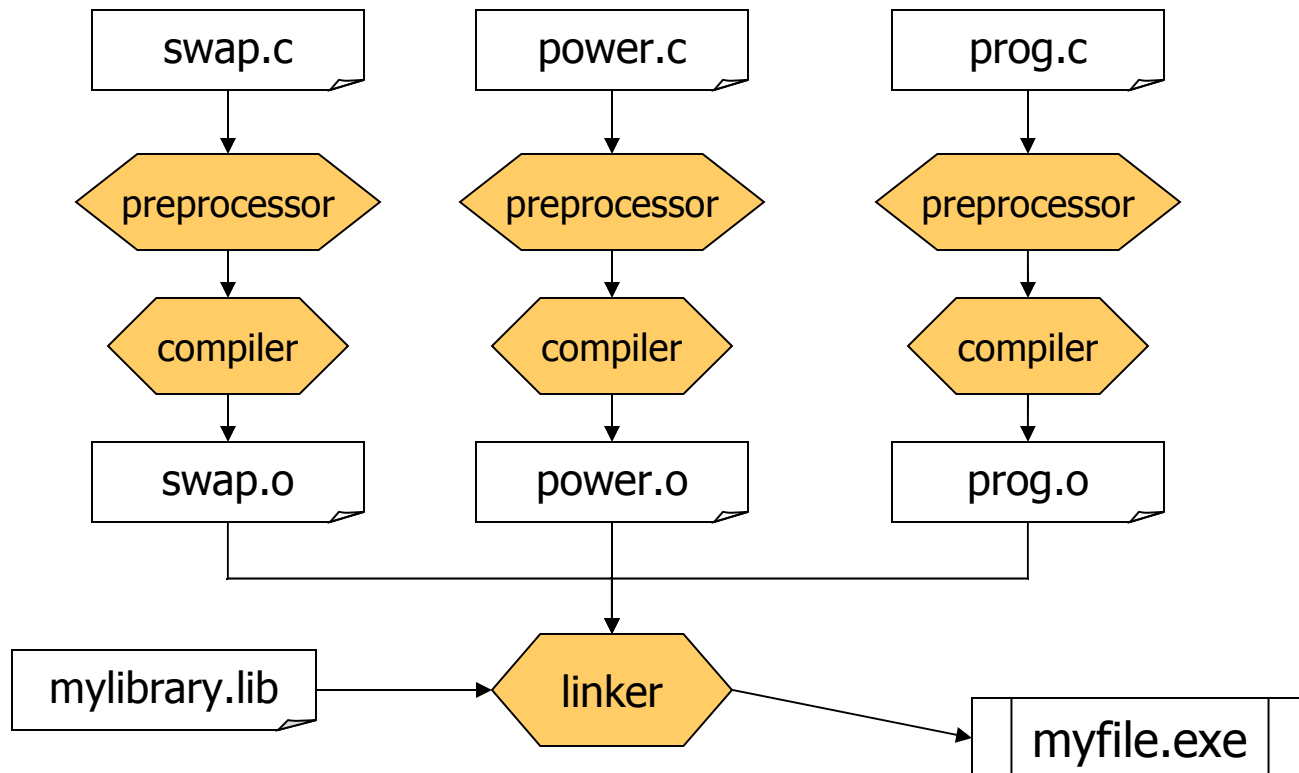
power.c

```
#include "power.h"  
  
double power(double x, int y)  
{  
    ...  
}
```



From sources to executable: update

- Compilation of each module can be done separately
- Results are then “linked” together by the linker





Visibility, scope, lifetime



Global and local variables

- Global variables (and other objects) are variables defined outside of any function
- Local variables (to a function) are defined inside a function
- The visibility (or scope) of a variable is the set of statements that can “see” the variable
 - remember that a variable (or any other object) must be declared before it can be used
- The lifetime of a variable is the time during which the variable exists in memory



Examples

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

```
/* array of prime numbers */  
int pn[100];
```

```
int is_prime(int x)  
{  
    int i, j;  
    ...  
}
```

```
int temp;
```

```
int main()  
{  
    int res;  
    char s[10];  
    ...  
}
```

Global variable (array)
Scope: all program
Lifetime: duration of the program

parameter
Scope: body of is_prime()
Lifetime: during execution of the function

local variables
Scope: body of is_prime()
Lifetime: during execution of the function

Global variable
Scope: function main
Lifetime: duration of the program

local variables
Scope: body of main
Lifetime: duration of the program



Hiding

- It is possible to declare two variables with the same names in two different scopes
- The compiler knows which variable to use depending on which scope the program is in
- It is also possible to “hide” a variable!

```
int fun1()
{
    int i;
    ...
}

int fun2()
{
    int i;
    ...
    i++;
}
```

increments the local variable of fun2

```
int i;

int fun1()
{
    int i;
    i++;
}

int fun2()
{
    i++;
}
```

increments the local variable of fun1

increments the global variable



Structures



Structures

- A structure is a set of non-homogeneous variables
- used to put together something that has to do with a certain world object
- Each data in the structure is called *field*
- The structure is sometime called *record*

```
struct student {  
    char name[20];  
    char surname[30];  
    int age;  
    int marks[20];  
    char address[100];  
    char country[100];  
};
```

```
struct student s1;
```

```
struct position {  
    double x;  
    double y;  
    double z;  
};  
  
struct position p1, p2, p3;
```



Accessing data

- To access a certain field of the structure, use the *dot* notation

```
struct student s1;  
...  
printf("Name: %s\n", s1.name);  
printf("Age: %d\n", s1.age);
```

```
#include <math.h>  
  
struct position p1;  
...  
p1.x = 10 * cos(0.74);  
p1.y = 10 * sin(0.74);
```



Arrays of structures

- It is possible to declare arrays of structures as follows

```
struct student my_students[20];
int i;

my_student[0].name = "...";
my_student[0].age = "...";
...

for (i=0; i<20; i++) {
    printf("Student %d\n", i);
    printf("Name: %s\n", my_student[i].name);
    printf("Age: %d\n", my_student[i].age);
    ...
}
```



More on Input/Output : Files



The concept of file

- A file is a sequence of bytes, usually stored on a mass-storage device
 - we can read or write a file sequentially (as on a magnetic tape)
- Files can contain
 - text (sequence of characters)
 - binary data
- In Unix there is not difference between text and binary files



file operations

- Before operating on a file, we must *open* it
- then we can operate on it (read or write)
- finally, we have to *close* the file when we have done
- In a C program, an open file is identified by a variable of type FILE



example of reading from a file

```
#include <stdio.h>

FILE *myfile;

int main()
{
    int a, b, c;
    char str[100];

    myfile = fopen("textfile.txt", "r");

    fscanf(myfile, "%d %d", &a, &b);
    fscanf(myfile, "%s", str);
    fscanf(myfile, "%d", &c);

    printf("what I have read:\n");
    printf("a = %d    b = %d    c = %d\n", a, b, c);
    printf("str = %s\n", str);
}
```



example of writing to a text file

```
#include <stdio.h>

FILE *myfile1;
FILE *myfile2;

int main()
{
    int i, nlines = 0;
    char str[255];

    myfile1 = fopen("textfile.txt", "r");
    myfile2 = fopen("copyfile.txt", "w");
    fgets(str, 255, myfile1);

    while (!feof(myfile1) {
        fprintf(myfile2, "%s", str);
        nlines++;
        fgets(str, 255, myfile1);
    }
    printf("file has been copied!");
    printf("%d lines read", nlines);
}
```