Programming and Algorithms

C: Functions.

Jean-Bernard Hayet

CIMAT, A.C.

Functions

Functions

- Code blocks that can call one to the other (including recursively).
- They may **return a value or not**.
- A program should have at least one: **the main function**.
- It contains local variables declaration and definition, instruction and return.

```
tipoX function(tipoY arg1, tipoZ arg2, ...) {
        tipoX t;
         ...
         return t;
}
```

Functions

```
int f(unsigned int n) {
        int result = 1;
        for (int k =1;k<=n ; k++)
                result *= k;
        return result ;
}
```

Functions

Caution: When calling the function, the order of the evaluation of the arguments is not necessarily from left to right, this is not specified by the standard.

```
f(a,a++,a-=2);
```

Functions

A function that does not return anything has the **void** type.

You can still use the return keyword (or not).

```
void g(unsigned int n) {
        printf("%d \n", f(n));
}
```

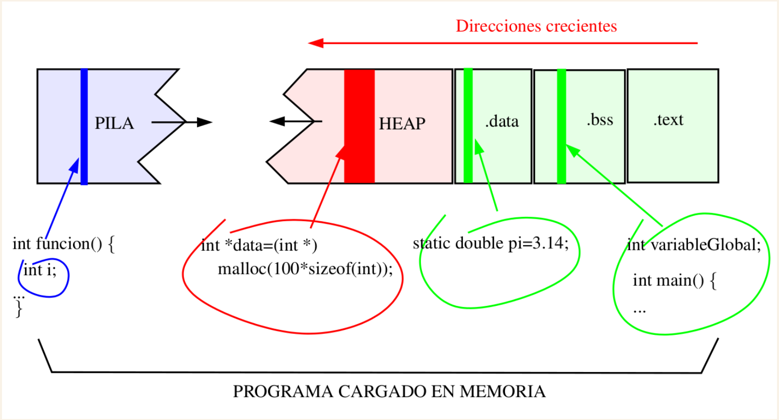
Functions: Memory layout

Memory resources in each function are **local**: They disappear when leaving the function.

All this memory is managed as a **stack**.

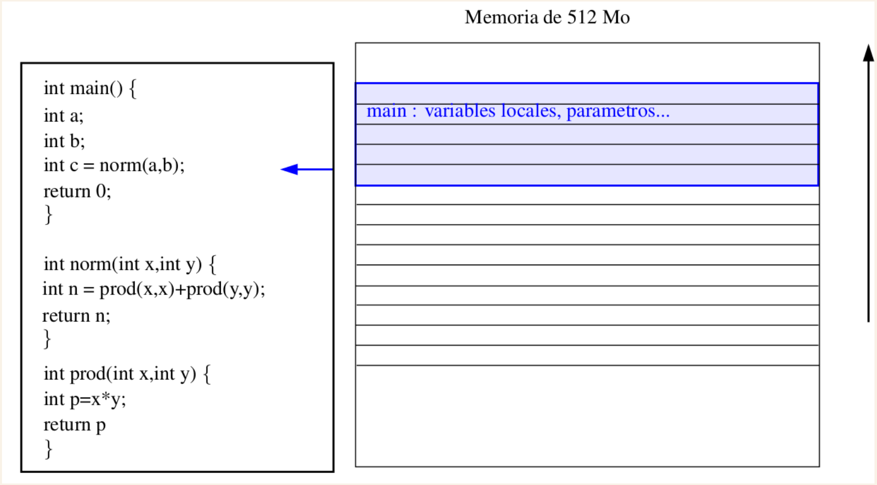
Each new call **uses the space left by the function that returned the last time**.

Functions: Memory layout



```
jbhayet@Barkoxe:~$ size a.out
text            data            bss             dec             hex         filenam
1540             600              8            2148             864         a.out
```

Functions

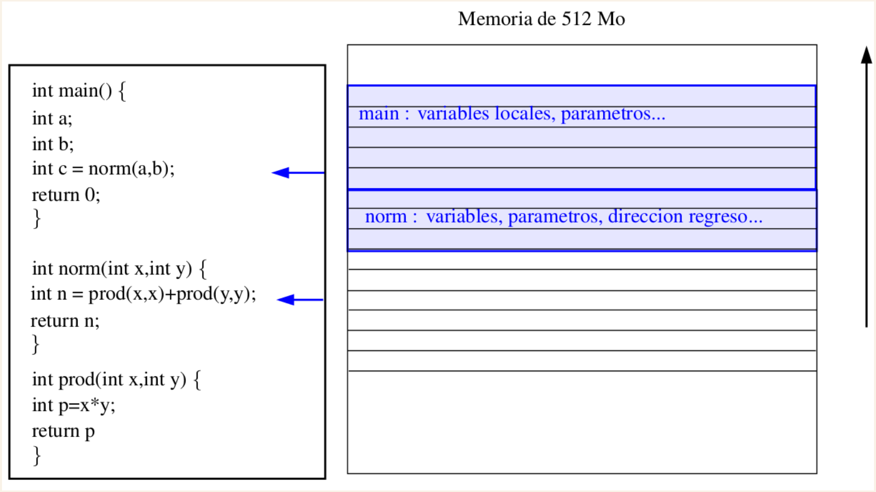


```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```

Functions



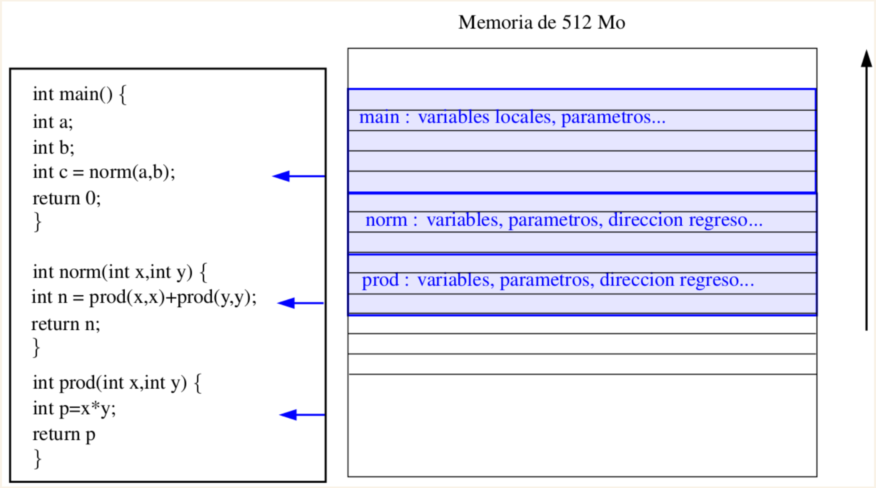
```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```

Memoria de 512 Mo
main : variables locales, parametros...
norm : variables, parametros, direccion regreso...

Functions



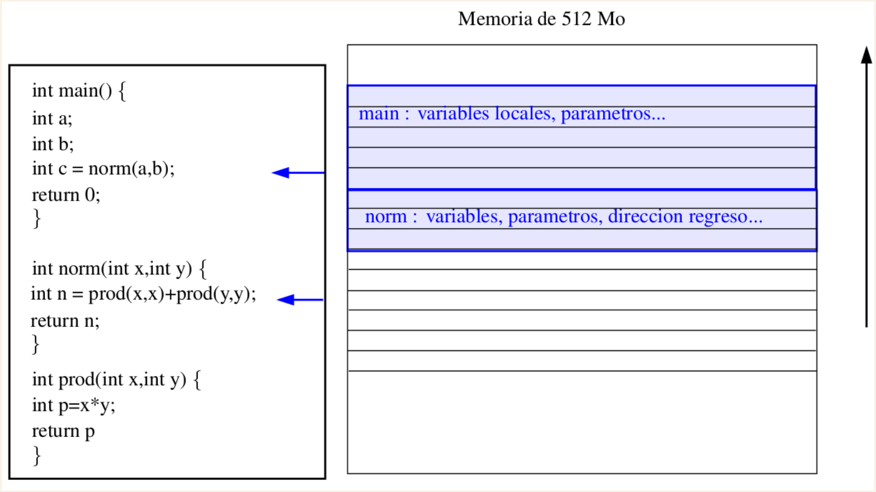
```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```

Memoria de 512 Mo
main : variables locales, parametros...
norm : variables, parametros, direccion regreso...
prod : variables, parametros, direccion regreso...

Functions



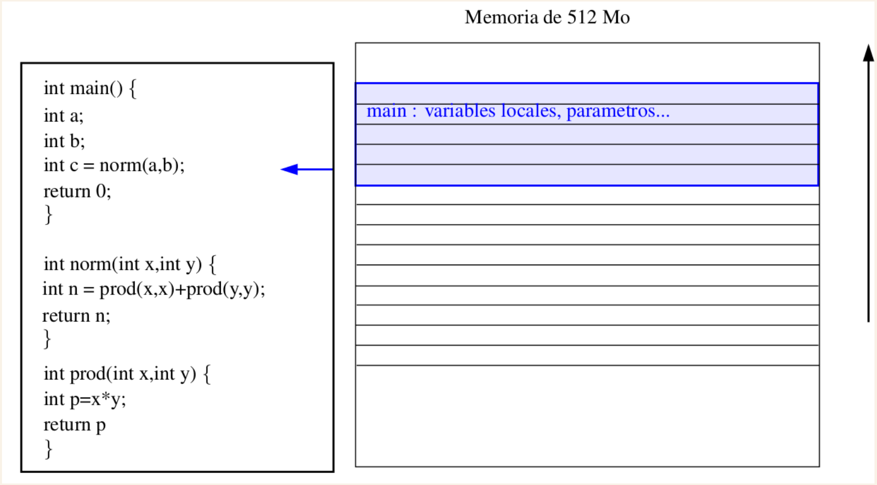
```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```

Memoria de 512 Mo
main : variables locales, parametros...
norm : variables, parametros, direccion regreso...

Functions



```
int main() {
int a;
int b;
int c = norm(a,b);
return 0;
}

int norm(int x,int y) {
int n = prod(x,x)+prod(y,y);
return n;
}

int prod(int x,int y) {
int p=x*y;
return p
}
```

Functions: The stack

When a function is called, **data are saved on the stack**:

- the address of where to return to;
- the memory address where the returned value will be stored;
- the arguments;
- information (registers) for the caller to start again.

Then the local (temporary) variables are then also allocated automatically on the stack.

“Stack **frames**”: No interaction between the context different function calls.

Functions: Stack vs. Heap

+ Access to stack memory is **faster** than the heap.
+ Memory is **less dispersed** (fragmentation), and allows a better use of the cache.
- **Less flexible**: memory in the heap can be modified (realloc function).

Functions: The stack

```
int f (unsigned int n) {
        if (n==0)
                return 1;
        else
                return n*f(n-1);
}
```

Recursive functions give nice code but they may imply some **memory overhead**!

Functions: The stack

The **stack size** is limited at the moment of running the program (8M in Linux by default.)

As many other hyperparameters on how a program runs c**an be consulted and even modified**:

```
> ulimit -a
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
stack size              (kbytes, -s) 8192
> ulimit -s kBytes
```

Functions: Static variables

Static variables are not stored following the stack mechanism.

Their visibility is **local** but their memory space is kept throughout calls.

```
void main() {
        static int k =1000; // .data
        static int l ;       // .bss
        int c = 10;          // Stack
}
```

```
jbhayet@Barkoxe:\~{}\$ size a.out
text          data          bss           dec           hex       filenam
1415           548           12          1975           7b7       a.out
```

Functions: Static variables

```
void main() {
 static int k;        // .data
 static int l;        // .bss
 int c = 10;           // Stack
}
```

```
jbhayet@Barkoxe:\~{}\$ size a.out
text           data           bss            dec            hex        filenam
1415            544            16           1975            7b7        a.out
```

Functions: Global variables

Some variables are defined out of any function: **Global variables**, visible in all the code that follows their declaration.

```
int nGlobal;
void f() {
        printf("Value of nGlobal %d\n", nGlobal++);
        return;
}
int main(){
        nGlobal =19;
        for (int k=0;k<10;k++)
                f();
        return 0;
}
```

Initialization at 0.

Functions: Name visibility

Caution to potential **name conflicts**.

```
int n;
void f() {
        int n =0;
        n = 18;
        ...
        return ;
}
```

Functions: Declarations

- Can't define a function within another (as in Python).
- A function that calls another needs to now the **prototype** of the function it calls.
- This implies that the **declaration** should come before.

```
int f(unsigned int n);
// Declaration of f
void g(unsigned int n) { // Definition of g
        printf( "%d\n",f(n)) ;
}
int f(unsigned int n) { // Definition of f
        int p=(n ==0)?1:n*f( n-1);
        return p ;
}
```

Functions: Declarations

To avoid problems with the order of functions declarations/definitions, it is strongly encouraged to **write all the declarations of your functions in a separate header file**, and keep the definitions in the .c

Content of func.h :

```
int  f(unsigned int n);  // Declaration of f
void g(unsigned int n); // Declaration of g
```

Functions: Declarations

Content of func.c, that includes the content of func.h (preprocessor) :

```
#include "func.h"
void g(unsigned int n) { // Definition of g
        ...
}
int f(unsigned int n) {
        // Definition of f
        ...
}
```

Functions: Declarations

Without the declaration, the compiler does not see an error. However:

```
#include <stdio.h>
// void h (int m,int n);
int main() {
        float b=6, c =9;
        h(b,c);
}
void h(int m,int n) { // Definition of h
        printf("%d %d\n",m,n) ;
}
```

Functions: Declarations

```
test2.c: In function ‘main’:
test2.c:5:2: warning: implicit declaration of function ‘h’ [-Wimplicit-function
h(b,c);
^
test2.c: At top level:
test2.c:7:6: warning: conflicting types for ‘h’
void h(int m,int n) { // Definition of h
^
test2.c:5:2: note: previous implicit declaration of ‘h’ was here
h(b,c);
^
```

At running:

1 -834978200

Functions: Declarations of external functions

One can use functions that are **defined elsewhere**, in external libraries; in that case it needs to be declared, with the keyword extern:

```
extern int putchar(int);
```

It means that when linking, the compiler should find somewhere the definition of this function, corresponding to this prototype.

Functions: Parameters

Parameters are processed as local variables: When the function is called, they are copied into the stack frame corresponding to the function call.

Then at leaving the function, these copies do not exist anymore.

```
void h (double x) {
        x =1.0;
        return ;
};
double y = 3.0;
h(y);
```

Functions: Passing variables through pointers

To use the parameters in writing mode, one needs to **pass a pointer to this variable as the parameter**.

```
void h(double *x) {
        *x =1.0;
};
double y = 3.0;
h(&y);
```

Functions: Error values

A common practice is to use the **return value for indicating errors**, while other "returned" data go through pointers as parameters:

```
int h(double *x) {
        *x =1.0;
        ...
        if (problem)
                return -1;
        return 0;
};
double y = 3.0;
if (!h(&y)) {
        ...
}
```

Functions: Error values

A particular case is the integer that is returned from main:

- 0 (EXIT SUCCESS, macro from stdlib.h) means that everything was OK,
- Non-zero values correspond to problems during execution (as EXIT FAILURE).
- Alternatively (system call):

```
exit ( int  status );
```

Pointers to functions

Pointers to functions

It can be useful to write code in terms of some functionalities but such that **the algorithm to implement the functionality can be changed**, without rewriting all the code.

Example: imagine a function that manipulates data and sort them. There are many algorithm to sort, so it could be cool to make an abstraction of the algorithm (see the function to use as a "variable").

Pointers to functions

A **pointer to function** is an object that allows to do this: It is a pointer (i.e. it corresponds to a **link to memory**). The pointed object is the machine code of function.

For functions with the prototype:

```
tipo funcion ( tipo1 ,... , tipon );
```

A pointer to functions with this prototype is described by:

```
tipo (*)( tipo1 ,... , tipon );
```

Pointers to functions

Caution:

```
tipo *func (tipo1 ,..., tipon ); // Declaration of a function
tipo (*func) (tipo1 ,..., tipon ); // Declaration of a pointer to functi
```

Can be **called** as:

```
func(v1 ,..., vn );
(*func)(v1 ,..., vn );
```

Pointers to functions: Example

```
int applySort(int *, int , int (*)(int *, int ));
```

A **function** that takes as arguments: a pointer to int, a number of data, and a **pointer to a sorting function** (i.e. a function that takes an array and a number of data and returning int).

Define:

```
int quickSort(int *data, int ndata) {
        ...
};
int heapSort(int *data, int ndata) {
        ...
};
```

Pointers to functions: Example

Could be used as:

```
int *data ;
int ndata ;
...
int err = applySort(data,ndata,quickSort );
```

Functions with variable number of arguments

Variable number of arguments

It is possible to define functions with a **variable number of arguments**. In that case, the prototype should have **a least one parameter specified**, and the list of optional parameters is represented by ...:

```
int varf(int a,char c ,...);
```

For example the prototype of printf:

```
int printf(char *format ,...);
```

Variable number of arguments

To acess the parameters, macros defined in stdarg.h

```
int sum(int ,...);
int sum(int num,...) {
        int res = 0;
        va_list listParams;
        va_start(listParams,num);
        for(int i=0;i<num;i++)
                res += va_arg(listParams,int);
        va_end(listParams);
        return(res) ;
}
```