Object Oriented Software Design

The C language

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

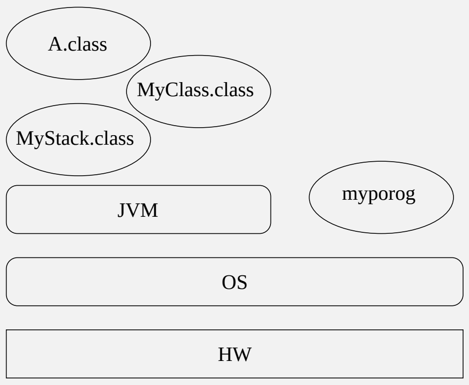
Scuola Superiore Sant'Anna – Pisa

November 15, 2010

Outline

The C language

- C++ is an object oriented language built upon C
  - Before looking at C++, let's have a quick look at how a C program is structured
- The C language is a compiled language
  - The C source code is transformed into an executable program
  - Unlike a Java compiler program (i.e. a set of .class files which needs a Java Virtual Machine), an executable file can be executed directly by the OS
  - This means that an executable program is not portable



Hello world in C

- 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

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)

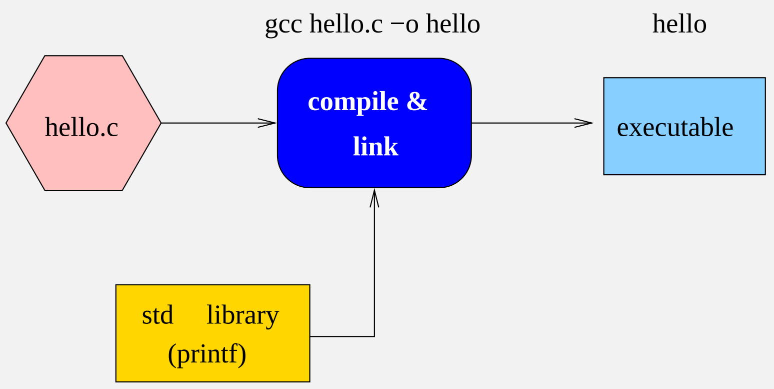
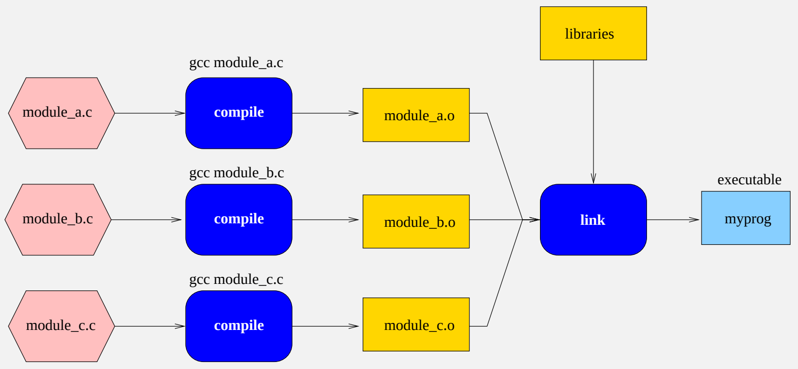


Figure: Compiling a file

Compiling a C program

- A C program consists of one ore more source files, each one is called a *compilation unit* or *module*
- Each unit is compiled separately, and a *object file* is generated as a result
- All objects files and the libraries are *linked* together to produce the executable file



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
  - Examples:

```
int a;             // declaration + definition
int b = 10;        // declaration + definition + init
extern int c;      // only declaration (no definition)
...
int c;             // definition

int f(int);        // only declaration

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

int g()            // declaration + definition
{

}
```

extern

- Keyword `extern` is used to specify that we are declaring something without defining it
- It is implicit for functions
- A function declaration is also called *function prototype*

Difference from Java

Notice that in Java every declaration is also a definition

Statements and expressions

- The Java syntax is a derivative of the C syntax
  - Therefore, in C you find similar statements to the ones you have already found in Java
  - `for(init ; cond ; expr) statement;`
  - `while (cond) statement;`
  - `if (cond) statement; else statement;`
  - `do statement while (cond);`
  - `switch (c) { case a :  statement; case b : statement; }`

Arrays

- Instead of single variables, we can declare arrays of variables of the same type
- They have all the same type and the same name
- They can be addressed by using an index

```
int i;
int a[10];

a[0] = 10;
a[1] = 20;
i = 5;
a[i] = a[i-1] + a[i+1];
```

- **Very important:** If the array has N elements, index starts at 0, and last element is at N-1
- In the above example, last valid element is `a[9]`

Example

dice.c

```
#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;
}
```

Index range

- What happens if you specify an index outside the array boundaries?

- The compiler does not complain, but you can get a random run-time error!
- Consider the following program: what will happen?

outbound.c

```
#include <stdio.h>

int main()
{
    int i;
    int a[10];

    for (i=0; i<15; i++) {
        a[i] = 0;
        printf("a[%d] = %d\n", i, a[i]);
    }

    printf("Initialization completed!\n");

    return 0;
}
```

Questions

- Index out of bounds is a programming error
  - Why the compiler does not complain?
  - Why the program does not complain at run-time?
- What is the memory allocation of the program? Where is the array allocated?

Initialization

- Arrays can be initialized with the following syntax

```
int a[4] = {0, 1, 2, 3};
```

- This syntax is only for static initialization, and cannot be used for assignment

```
int a[4];

a = {0, 1, 2, 3};  // syntax error!
```

Matrix

- Two- and three-dimensional arrays (matrices):

```
double mat[3][3];
int cube[4][4][4];

mat[0][2] = 3.5;
```

- Static and dynamic initialisation

matrix.c

```
#include <stdio.h>

int main()
{
    int i;
    double mat[3][3] = {
        {0, 0, 0},
        {0, 0, 0},
        {0, 0, 0}
    };
    mat[0][2] = 3.5;
    for (i=0; i<9; i++) {
        mat[i/3][i%3] = 2.0;
    }
    printf("Done\n");
    return 0;
}
```

Structure definition

- In many cases we need to aggregate variables of different types that are related to the same concept
- each variable in the structure is called *a field*
- the structure is sometimes called *record*
- Example

```
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 field of a structure, use the *dot notation*

```
#include <math.h>

struct position {
  double x;
  double y;
  double z;
};

struct position p1;
...
p1.x = 10 * cos(0.74);
p1.y = 10 * sin(0.74);
```

Function definition and declaration

- A function is defined by:
  - a unique name
  - a return value
  - a list of arguments (also called parameters)
  - a body enclosed in curly braces

- An example: this function raises a double number to an integer power

```
/* 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;
}
```

Function call

- This is how the function is called.
- The formal parameters x and y are substituted by the actual parameters (the values of xx and yy)

power.c

```
int main()
{
    double myx;
    int myy;
    double res;

    printf("Enter x and y\n");
    printf("x? ");
    scanf("%lg", &myx);
    printf("y? ");
    scanf("%d", &myy);

    res = power(myx, myy);

    printf("x^y = %lgt\n", res);
}
```

Parameters

- Modifications on local parameters have no effect on the caller

```
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
- modifying x modifies the copy, **not** the original value
- We say that in C parameters are passed *by value*
- There is only one exception to this rule: arrays
  - An array parameter is never copied, so modification to the local parameter are immediately reflected to the original array

Array parameters

swap.c

```
#include <stdio.h>

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 initialised when declared

Definitions

- **Global variables** are variables defined outside of any function
- **Local variables** 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>

int pn[100];

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

int temp;

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

pn is a global variable
scope: all program
lifetime: duration of the program

x is a parameter
scope: body of function is_prime
lifetime: during function execution

i,j are local variables
scope: body of function is_prime
lifetime: during function execution

temp is a global variable
scope: all objects defined after temp
lifetime: duration of the program

res and s[] are local variables
scope: body of function main
lifetime: duration of the program

Global scope

- A **global variable** is declared outside all functions
  - This variable is created before the program starts executing, and it exists until the program terminates
  - Hence, it's **lifetime** is the program duration
- The **scope** depends on the point in which it is declared
  - All variables and functions defined after the declaration can use it
  - Hence, it's scope depends on the position

Local variables

- Local variables are defined inside functions

```
int g;

int myfun()
{
   int k; double a;
   ...
}

int yourfun()
{
   ...
}
```

g is global

k and a are local to myfun()

in function yourfun(), it is possible to use variable g but you cannot use variable k and a (out of scope)

- k and a cannot be used in `yourfun()` because their scope is limited to function `myfun()`.

Local variable lifetime

- Local variable are *created* only when the function is invoked;
- They are *destroyed* when the function terminates
  - Their lifetime corresponds to the function execution
  - Since they are created at every function call, they hold only temporary values useful for calculations

```
int fun(int x)
{
   int i = 0;

   i += x;
   return i;
}

int main()
{
   int a, b;

   a = fun(5);
   b = fun(6);

   ...
```

i is initialized to 0 at every fun() call

at this point, a is 5 and b is 6;

Modifying lifetime

- To modify the lifetime of a local variable, use the `static` keyword

```
int myfun()
{
    static int i = 0;

    i++;

    return i;
}

int main()
{
    printf("%d ", myfun());
    printf("%d ", myfun());
}
```

This is a static variable: it is initialised only once (during the first call), then the value is maintained across successive calls

This prints 1

This prints 2

Hiding

- It is possible to define two variables with the same name in two different scopes
- The compiler knows which variable to use depending on the scope
- 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

Pre-processor

- In the first step, the input file is analyzed to process *preprocessor directives*
- A preprocessor directive starts with symbol #
  - Example are: **`#include`** and **`#define`**
- After this step, a (temporary) file is created that is then processed by the compiler

Directives

- With the **include** directive, a file is included in the current text file
  - In other words, it is copied and pasted in the place where the include directive is stated
- With the **define** directive, a symbol is defined
  - Whenever the preprocessor reads the symbol, it substitutes it with its definition
  - It is also possible to create macros
- To see the output of the pre-processor, run gcc with -E option (it will output on the screen)

```
gcc -E myfile.c
```

An example

main.c

```
#include "myfile.h"
#include "yourfile.h"

int d;
int a=5;
int b=6;

int main()
{
    double c = PI;    // pi grego
    d = MYCONST;      // a constant
    a = SUM(b,d);     // a macro
    return (int)a;
}
```

myfile.h

```
#define MYCONST 76
extern int a, b;
#define SUM(x,y) x+y
```

yourfile.h

```
#define PI 3.14
extern int d;
```

main.c.post

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"
# 1 "myfile.h" 1


extern int a, b;
# 2 "main.c" 2
# 1 "yourfile.h" 1


extern int d;
# 3 "main.c" 2

int d;
int a=5;
int b=6;

int main()
{
    double c = 3.14;
    d = 76;
    a = b+d;
    return (int)a;
}
```

Macros effects

- Pay attention to macros, they can have bad effects

```
#define SUM(x,y) x+y

int main()
{
  int a = 5, b = 6, c;

  c = 5 * SUM(a,b);
}
```

- What is the value of variable c?

Some helpful “tricks”

- It is possible to define a macro for obtaining the literal name of a variable:

```
#define LIT_VAR(x) #x
```

A complete example:

point2.c

```
#include <stdio.h>

#define LIT_VAR(a) #a
#define PVAR(y) printf("%s = %d", LIT_VAR(y), y)
#define PPUN(y) printf("%s = %p", LIT_VAR(y), y)

int main()
{
    int d = 5;
    int x = 7;
    int *pi;

    pi = &x;

    PVAR(d);  PPUN(&d);
    PVAR(x);  PPUN(&x);
    PPUN(pi); PVAR(*pi);

    d = *pi;

    PPUN(pi); PVAR(x);
    PVAR(d);
}
```

Include files

- Include files are used to declare the module **interface**
  - they contain all declarations that the module wants to export to other modules
- An include file should not contain definitions, but only declarations!
  - In fact, suppose an include file `myfile.h` contains the definition of a variable `int a;`
  - Now suppose that the file is included by two modules, `ma.c` and `mb.c`
  - When compiling `ma.c`, an integer variable is created in memory and is called a;
  - When compiling `mb.c`, another integer variable is created in memory and is also called a;
  - **the linker** will try to put together `ma.o` and `mb.o` and will find two variables with the same name; it may give you an error!!
  - In any case this is an error!
- What you should do:
  - put the declaration `extern int a;` in `myfile.h`;
  - put the definition `int a;` in one file only, `ma.c` or `mb.c`

Summary of differences between Java and C

Java

- Portable programs
- Declaration and definition coincide (no need for include files)
- There is no global scope, all variables and functions are defined inside classes
- It is not possible to hide a variable
- Array bounds are checked at run-time and an exception is raised for index out of bound

C

- Non portable programs (must be recompiled)
- It is possible to declare a variable (or a function) and later define it (difference between .c and .h)
- Variables and functions can be in the *global scope*
- It is possible to hide a variable inside a scope
- There is no check at run time for array bounds

Pointers

- A pointer is a special type of variable that can hold *memory addresses*
- Syntax

```
char c;      // a char variable
char *pc;    // pointer to char variable
int i;       // an integer variable
int *pi;     // pointer to an int variable
double d;    // double variable
double *pd;  // pointer to a double variable
```

- In the declaration phase, the `*` symbol denotes that the variable contains the address of a variable of the corresponding type

Syntax - cont.

- A pointer variable may contain the address of another variable

```
int i;
int *pi;

pi = &i;
```

- The `&` operator is used to obtain the address of a variable.
- It is called the *reference* operator
  - Warning: in C++ a reference is a different thing! Right now, pay attention to the meaning of this operator in C.

Indirection

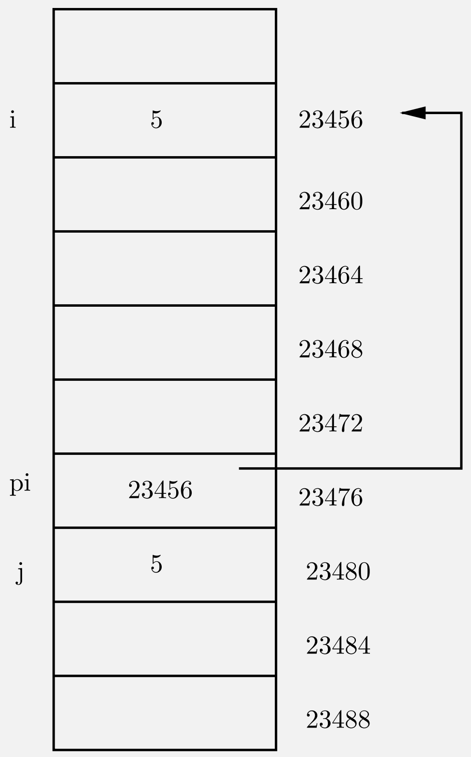
- The reverse is called *indirection* operator and it is denoted by `*`

```
int j;
j = *pi;  // get the value pointed by pi

*pi = 7;  // store a value in the address stored in pi
```

- In the first assignment, `j` is assigned the value present at the address pointed by `pi`.
- In the second assignment, the constant `7` is stored in the location contained in `pi`
- `*pi` is an *indirection*, in the sense that is the same as the variable whose address is in `pi`

Example

- `pi` is assigned the address of `j`
- `j` is assigned the value of the variable pointed by `pi`



Examples

point1.c

```
int main()
{
    int d = 5;
    int x = 7;
    int *pi;

    pi = &x;

    printf("%p\n", &x);
    printf("%p\n", &d);
    printf("%p\n", pi);

    printf("%d\n", *pi);

    //pi = d;  // compilation error

    d = *pi;

    printf("%p\n", pi);
    printf("%d\n", x);
    printf("%d\n", d);
}
```

The commented line is a syntax error

- We are assigning a variable to a pointer
- The programmer probably forgot a `&` or a `*`

Arguments of function

- In C, arguments are passed by value
  - With the exception of arrays
- However, we can use pointers to pass arguments by *reference*

```
void swap(int *a, int *b)
{
  int tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;
}

int main()
{
  int x = 1;
  int y = 2;

  swap(&x, &y);

  PVAR(x);
  PVAR(y);
}
```

Pointers and arrays

- An array denotes a set of consecutive locations in memory
- In C, the name of an array is seen as a *constant pointer* to the first location
- Therefore, it can be assigned to a pointer, and used as a pointer

```
int array[5] = {1, 2, 4, 6, 8};
int *p;
int d;

p = a;
d = *p;     // this expression has value 1
```

Pointer arithmetic

- It is possible to modify a pointer (i.e. the address) by incrementing/decrementing it

```
int a[5] = {1, 2, 3, 4, 5};
int *p;
p = a;      // p now points to the first
            // element in the array

p++;        // p now points to the second
            // element (a[1])

p+=2;       // p now points to the fourth
            // element (a[3])
```

- Notice that in `p++`, `p` is incremented by 4 bytes, because `p` is a pointer to integers (and an integer is stored in 4 bytes)

Array and pointers

- Array are constant pointers, they cannot be modified

```
int a[10];
int d;
int *p;

p = &d;

a = p;  // compilation error, a cannot be modified
```

- Remember that the name of an array is not a *variable*, but rather an address!
- It can be used in the right side of an assignment expression, but not in the left side.

Equivalent syntax

- A pointer can be used to access the elements of an array in different ways:

```
int a[10];
int *p;

p = a;
*(p+1);   // equivalent to a[1]

int i;

*(p+i);   // equivalent to a[i]
p[i];     // this is a valid syntax
*(a+i);   // this is also valid
```

- In other words, a and p are equivalent also from a syntactic point o view

Pointer arithmetic - II

- The number of bytes involved in a pointer operator depend on the pointer type
- An operation like `p++` increments the pointer by
  - 1 byte if p is of type **`char`**
  - 2 bytes if p is of type **`float`**
  - 4 bytes if p is of type **`int`**
- To obtain the size of a type, you can use the macro **`sizeof`**`()`

```
int a, b;
char c;
double d;

a = sizeof(int);  // a is 4 after the assignment
a = sizeof(c);    // c is a char, so a is assigned 1
```

- **`sizeof`**`()` must be resolved at compilation time (usually during preprocessing)

Pointer arithmetic - III

- Pointer arithmetic is also applied to user-defined types;

struct.c

```
#include <stdio.h>

typedef struct mystruct {
    int a;
    double b[5];
    char n[10];
};

int main()
{
    struct mystruct array[10];

    printf("size of mystruct: %ld\n", sizeof(struct mystruct));

    struct mystruct *p = array;

    printf("p = %p\n", p);
    p++;
    printf("p = %p\n", p);
}
```

void pointers

- In C/C++, the keyword **void** denotes something without a type
  - For example the return value of a function can be specified as void, to mean that we are not returning any value
- When we want to define a pointer that can point to a variable of any type, we specify it as a void pointer

```
void *p;
int d;

p = &d;
p++;      // error, cannot do arithmetic
          // with a void pointer
```

Pointers and structures

- When using pointers with structures, it is possible to use a special syntax to access the fields

```
struct point2D {
  double x, y;
  int z;
};

point2D vertex;
point2D *pv;    // pointer to the structure

pv = &vertex;
(*pv).x;       // the following two expressions
p->x;          // are equivalent
```

- Therefore, to access a field of the structure through a pointer, we can use the arrow notation `p->x`

Java vs C - II

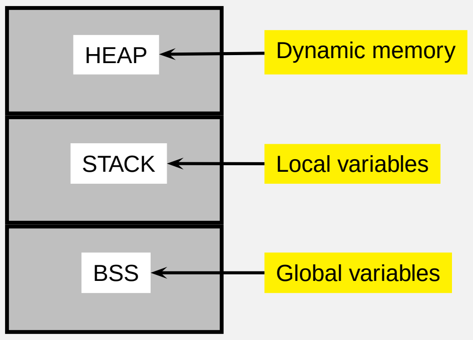
- There are no pointers in Java
  - Java references are similar to pointers
  - However, you cannot do arithmetic with references
  - Also, you cannot directly address memory in Java (except by using special OS interface, for example for accessing external devices)
- Pointers are low-level
  - They allow a C programmer to access memory directly
  - However, there is no run-time check on how the programmer uses them, for efficiency reasons
  - They can be the source of many difficult and subtle errors

Memory allocation

- We have discussed the rules for the lifetime and visibility of variables
  - **Global variables** are defined outside of any function. Their lifetime is the duration of the program: they are created when the program is loaded in memory, and deleted when the program exits
  - **Local variables** are defined inside functions or inside code blocks (delimited by curly braces { and }). Their lifetime is the execution of the block: they are created before the block starts executing, and destroyed when the block completes execution
- Global and local variables are in different **memory segments**, and are managed in different ways

Memory segments

- The main data segments of a program are shown below
- The BSS segment contains **global variables**. It is divided into two segments, one for initialised data (i.e. data that is initialised when declared), and non-initialised data.
  - The size of this segment is statically decided when the program is loaded in memory, and can never change during execution
- The STACK segment contains **local variables**
  - Its size is dynamic: it can grow or shrink, depending on how many local variables are in the current block



Example

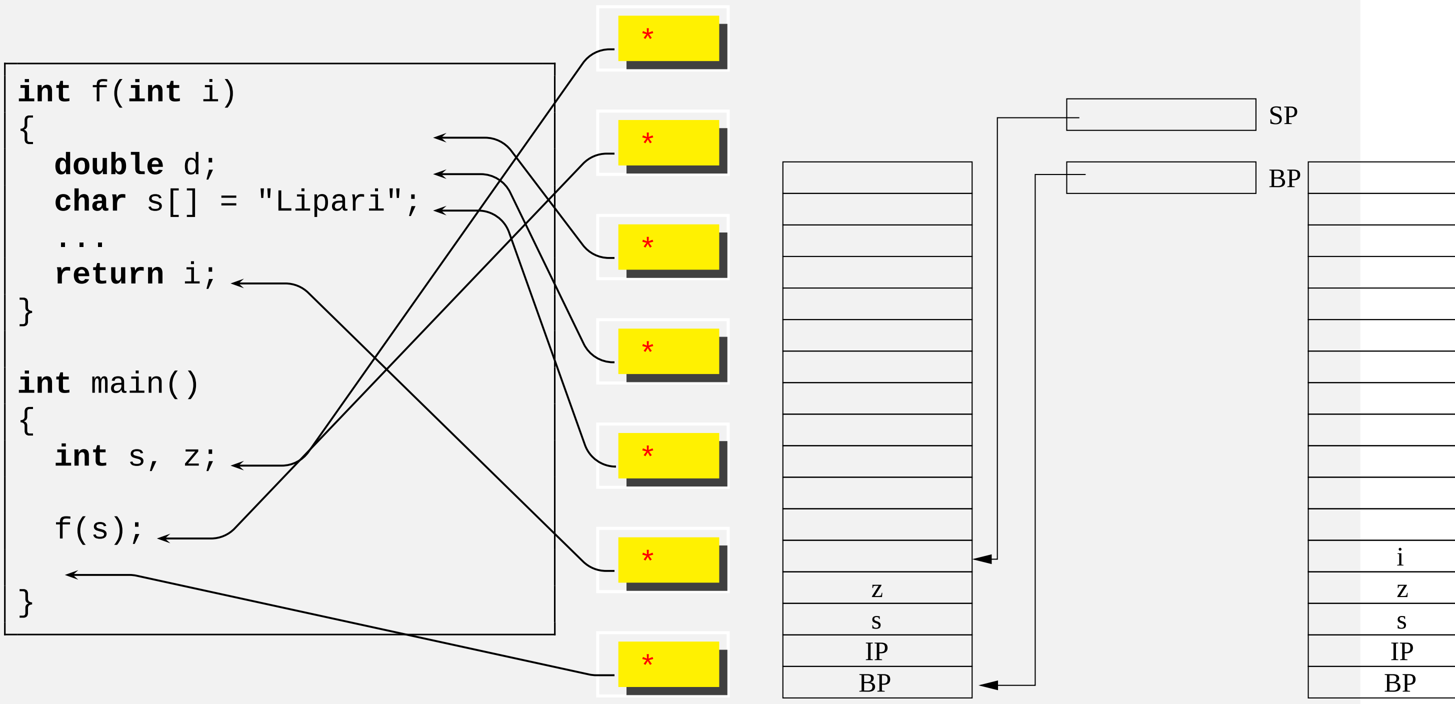
- Here is an example:

```
int a = 5; // initialised global data
int b;     // non initialised global data

int f(int i)    // i, d and s[] are local variables
{               // will be created on the stack when the
  double d;     // function f() is invoked
  char s[] = "Lipari";
  ...
}

int main()
{
  int s, z;     // local variables, are created on the stack
                // when the program starts

  f();          // here f() is invoked, so the stack for f() is created
}
```

Stack

- A Stack is a data structure with two operations
  - **push** data on top
  - **pop** data from top
- The stack is a LIFO (last-in-first-out) data structure
- The stack memory is managed in the same way as the data structure
- When a function is called, all parameters are **pushed** on to the stack, together with the local data
  - The set of function parameters, plus return address, plus local variables is called **Stack Frame** of the function
  - The CPU internally has two registers:
    - **SP** is a pointer to the top of the stack
    - **BP** is a pointer to the current *stack frame*
  - while the function is working, it uses **BP** to access local data
  - when the function finishes, all data is **popped** from the stack

Stack



```
int f(int i)
{
  double d;
  char s[] = "Lipari";
  ...
  return i;
}

int main()
{
  int s, z;

  f(s);

}
```

Stack frames

- Every time we call a function we generate a different stack frame
  - Every stack frame corresponds to an *instance* of the function
  - Every instance has its own variables, different from the other instances
- Stack frame is an essential tool of **any** programming language (including Java)