Informática Industrial

Práctica 1: Introducción Programación C++. Codeblocks

Del C al C++

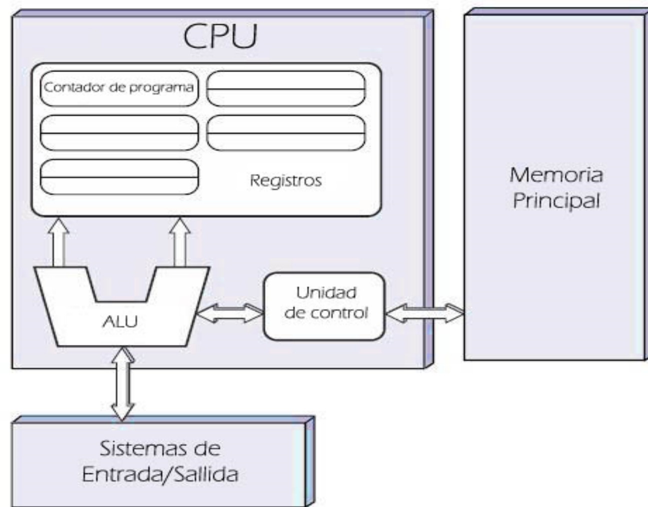
- Objetivos que nos proponemos
- Breve historia.
- Primer programa en C++ en CodeBlocks

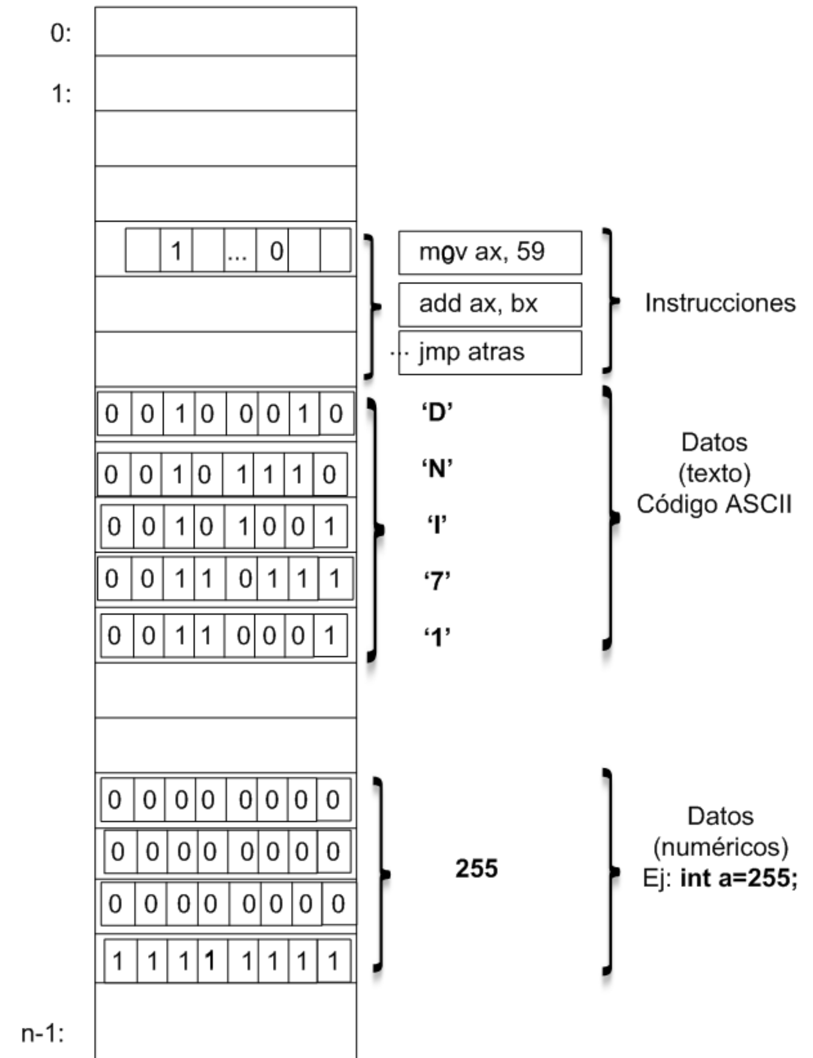
Del C al C++

- Objetivos:
  - No pretendemos que el alumno sea un "experto" en C++
  - Aprovechar los conocimientos de C para ir introduciendo conceptos más avanzados de Orientación a Objetos
    - Primero: Utilizar clases y objetos definidos en la líbrería estándar de C++
    - Finalmente: Ser capaces de diseñar y definir alguna clase sencilla propia
  - El trabajo con los conceptos y librerías de C++ nos permitirá trabajar a un nivel de abstracción mayor, lo que a la larga facilitará el aprendizaje de los principios propios de la asignatura.
  - El C++ ha experimentado una evolución considerable a partir de la versión 11 (C++v11). Hay otros estándar aprobado en vía de ser implementados en los compiladores (v14) y otro (v17) a punto de ser aprobado. Estudiaremos C++v11 lo que se ha dado en llamar "**C++ moderno**".

Del C al C++. Antecedentes

Arquitectura de Von Neumann





- Arquitectura **Von Neumann**:
  - **Bloques funcionales** (CPU, Memoria, E/S) conectada por ***buses*** de datos, direcciones y control
  - La CPU ejecuta instrucciones elementales en **código de máquina** (aritméticas, de e/s, de memoria, saltos en la ejecución del programa
  - **Memoria** contiene tanto **instrucciones** de el(los) proceso(s) a ejecutar como los **datos** (se determina qué es qué por el contexto)

Del C al C++. Antecedentes

- Evolución de los lenguajes de programación:
  - **Lenguajes de bajo nivel:**
    - **Lenguaje de máquina** (los "1" y "0") del conjunto de instrucciones específicos de la CPU de que se trate
    - **Lenguaje ensamblador**: emplea palabras nemotécnicas para refirirse a las misma instrucciones de código de máquina (más cómodo de usar)

Ensamblador	Lenguaje de máquina
`ADD AX,54`	`01000101` `01010100`

Ensamblador	Lenguaje de máquina

`atras:`
…
`jmp atras`
	`00000101`
	…
	`11010100`

- Son esencialmente **equivalentes**
- El **ensamblador** debe ser **traducido** a **lenguaje de máquina** para ser ejecutado
- La traducción es **simple** y unívoca

Lectura complementaria: http://www.learncpp.com/cpp-tutorial/02-introduction-to-programming-languages/

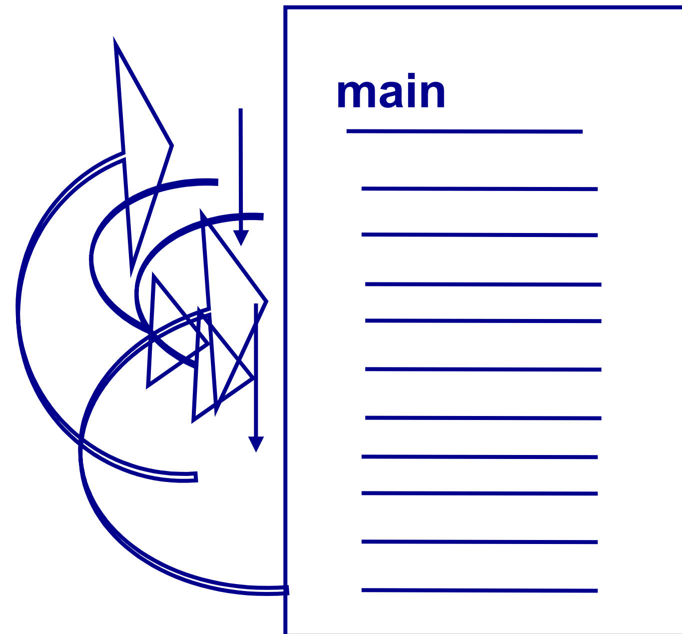
Del C al C++. Antecedentes

- Evolución de los lenguajes de programación:
  - **Lenguajes de alto nivel:**
    - El primero (FORTRAN) creado en la década de 1950.
    - Idea: redactar los programas como **fórmulas matemáticas** y que estas fueran traducidas a **código de máquina**.
    - La **correspondencia** ahora **no es sencilla ni unívoca**. Una misma sentencia de alto nivel puede ser traducida de diversas maneras.
    - El programa que hace la conversión (**compilador**) es **complejo**.
    - **Ejemplos** de lenguajes de alto nivel: FORTRAN, **C**, Pascal, Java, **C++**,...

Del C al C++. Antecedentes

- Evolución de los lenguajes de programación:
  - **¿Cómo programar bien?**
    - Rápidamente el uso de ordenadores tuvo un gran impacto económico
    - Las aplicaciones crecían en complejidad y las empresas gastaban muchos recursos y tiempo en mantener programas existentes.
    - Los primeros programas de alto nivel eran difíciles de mantener:
      - Aplicaciones monolíticas
      - se usaban muchos **goto** equivalentes a los **saltos** de lenguaje ensamblador

Código espagueti

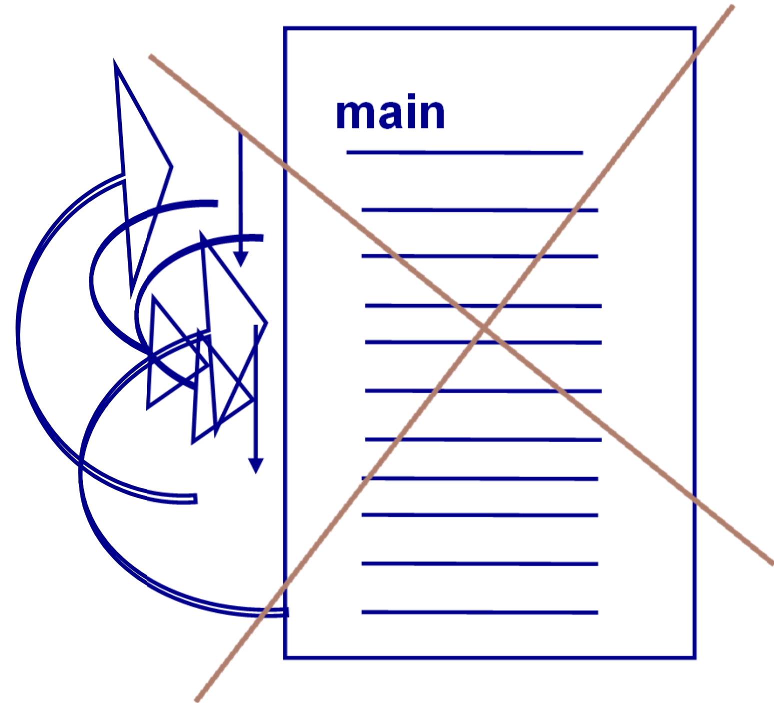


Del C al C++. Antecedentes

Evolución de los lenguajes de programación:

Se necesitaba nuevos **paradigmas** de programación: conjunto de **reglas** o teoría matemática que permita alcanzar el ideal de **programar "bien"**

Código espagueti

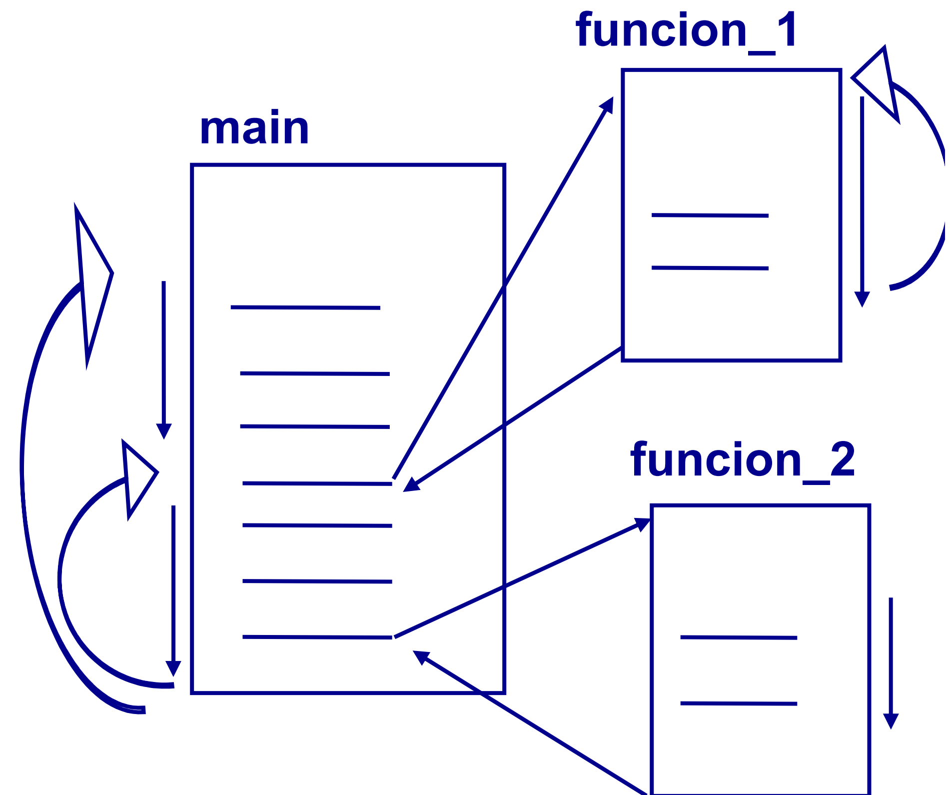


Del C al C++. Antecedentes

Evolución de los lenguajes de programación:

Paradigma de programación estructurada (C, Pascal, ...):

- El **goto** considerado dañino: siempre se podía sustituir por estructuras más claras:
  - secuenciales
  - iterativas
  - condicionales
- Descomponer la tarea en módulos
  - **Interfaz** bien definida
  - Cada módulo se dedica a una tarea concreta, (**cohesión**) bien definida y no depende de otros (**desacoplamiento**)
  - **Esconden código**
  - Mejoran la **depuración**
  - Permiten **colaboración** de equipos de programadores
  - Estimulan diseño **top-down**

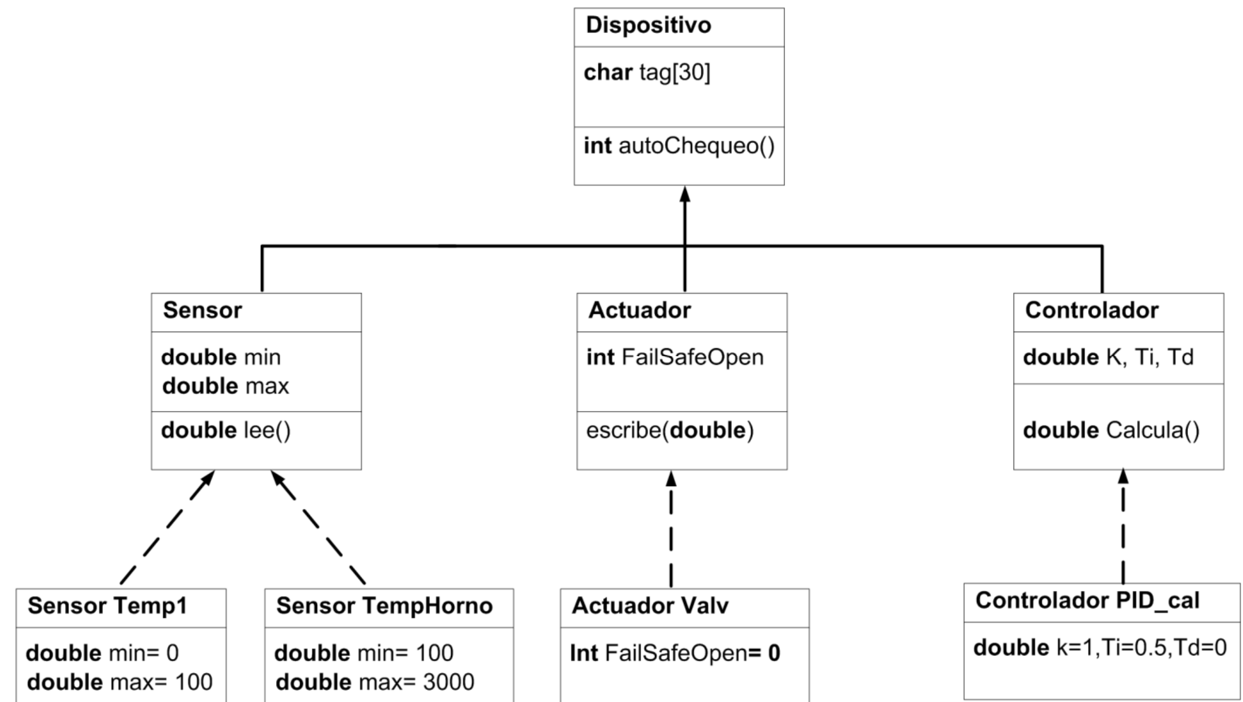


Del C al C++. Antecedentes

Evolución de los lenguajes de programación:

Paradigma de programación Orientada a Objetos (C++, Java, ...):

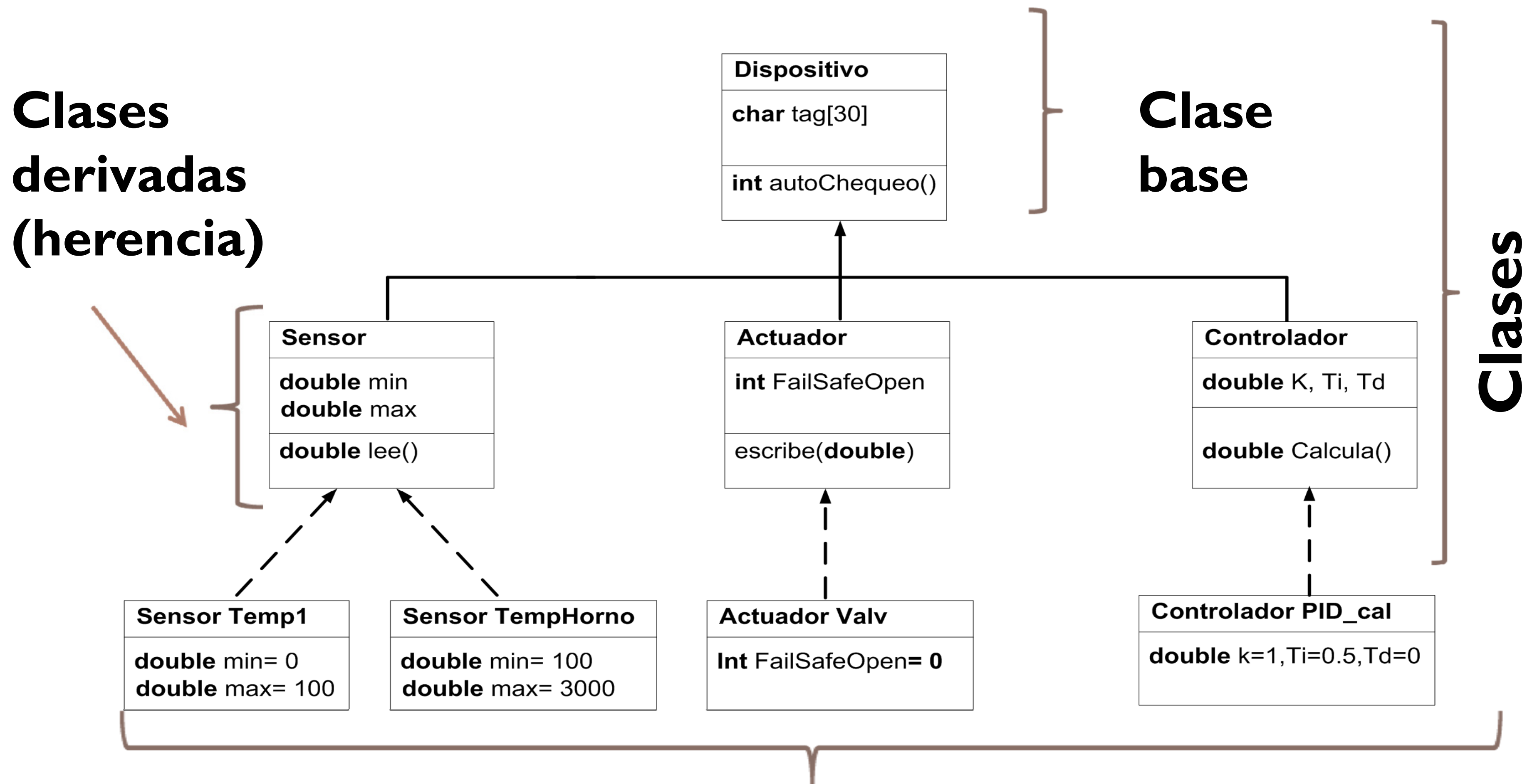
- El programa es concebido como un conjunto de **objetos** que continen **datos** y que exhiben un **comportamiento**.



- Promueven un diseño del programa de tipo bottom-up
- Las **clases** juegan el papel de los tipos de datos y los **objetos** (instancias) el de de las variables.
- Poseen propiedades de **herencia**, **encapsulamiento**, **polimorfismo**, **agregación**...

Del C al C++. Antecedentes

Paradigma Orientada a Objetos



Objetos (instancias de sus respectivas clases)

Del C al C++. Antecedentes

Comparación

Prog. estructurada (énfasis en ***verbos***)	Prog. Orientada a Objetos (énfasis en ***sustantivos***)
Lazo de control: se **lee** sensor de temp., se **determina** la señal del controlador y se **envía** a la válvula.	Lazo de control: se lee **sensor de temp.**, se determina la señal del **controlador** y se envía a la **válvula**.

double y , m;
 	
....
while (...)
{
  y= **lee_sensor**(“Temp1”);
  m=**calcula_control**(“PID_cal”, y);
  envia_actuador(“Valv”,m);
}
	sensor temp;
	controlador cont;
	actuador valvula;
	double y , m;
	....
	while (...)
	{
	  y= **temp**.lee();
	  m=**cont**. Calcula(y);
	  valvula.escribe(m);
	}

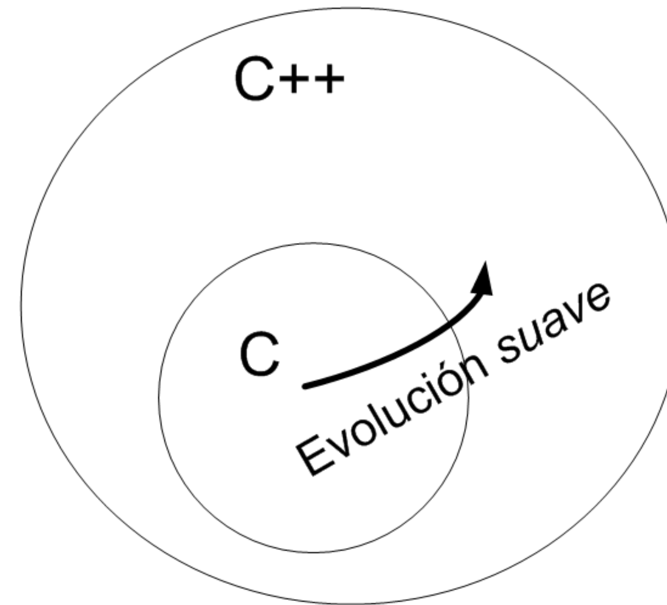
Del C al C++. Antecedentes

- Características:
  - Lenguaje C:
    - Creado en los laboratorios Bell (ATT) en los 1970s. Dennis Ritchie.
    - Lenguaje de alto nivel pero con recursos para "bajar" al *hardware*
    - Sintaxis que lo hacen apto para programación estructurada o modular
    - Estándar ANSI en los 90's
  - Lenguage C++:
    - Creado en laboratorios Bell en los 1990s. Bjarne Stroustroup.
    - Incluye el C como subconjunto
    - Permite implementar el paradigma Orientado a Objetos. No es el primer lenguaje OO (Simula, Smalltalk) pero si el que popularizó el concepto
    - El C++ moderno (de C++ v11 en adelante) ha incorporado muchos nuevos conceptos, incluyendo soporte nativo a la concurrencia.

  - Lectura complementaria http://www.learncpp.com/cpp-tutorial/03-introduction-to-cc/

Del C al C++. Estrategia de estudio de C++

- **Estrategia**:
  - Presentar las **novedades** en las medidas que nos hagan falta: Trabajar a un **mayor nivel de abstracción**: **mejorar productividad**
  - Introducir mejoras útiles de sintaxis no necesariamente del tipo OO
  - Aprender a **utilizar clases** ya definidas en la librería para **después crear** algunas propias
  - Tener una visión **complementaria**: algunas cosas se harán al estilo **procedural** y otras en **orientado a objetos**.



Del C al C++

Proceso de compilación

Programa fuente (*.cpp)

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Preprocesamiento

Modifica texto de acuerdo directivas del preprocesador

Programa fuente (*.i)

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

#pragma GCC system_header

#include <bits/c++config.h>
#include <ostream>
#include <istream>

_GLIBCXX_BEGIN_NAMESPACE(std)
```

Compilación

Convierte sintaxis de alto nivel en lenguaje de máquina

Programa objeto (*.obj)

```
11010101010101010101
0100101010101000010 1
01010100110101001 0
10100010101010110 0
```

Librerías (*.lib)
Otros (*.obj)

```
11010101010101010101
01001010101010000101
01010100110101001 0
10100010101010110 0
```

Enlazador (Linker)

Crea el programa a ejecutar uniendo código pre-compilado (ej: librerías)

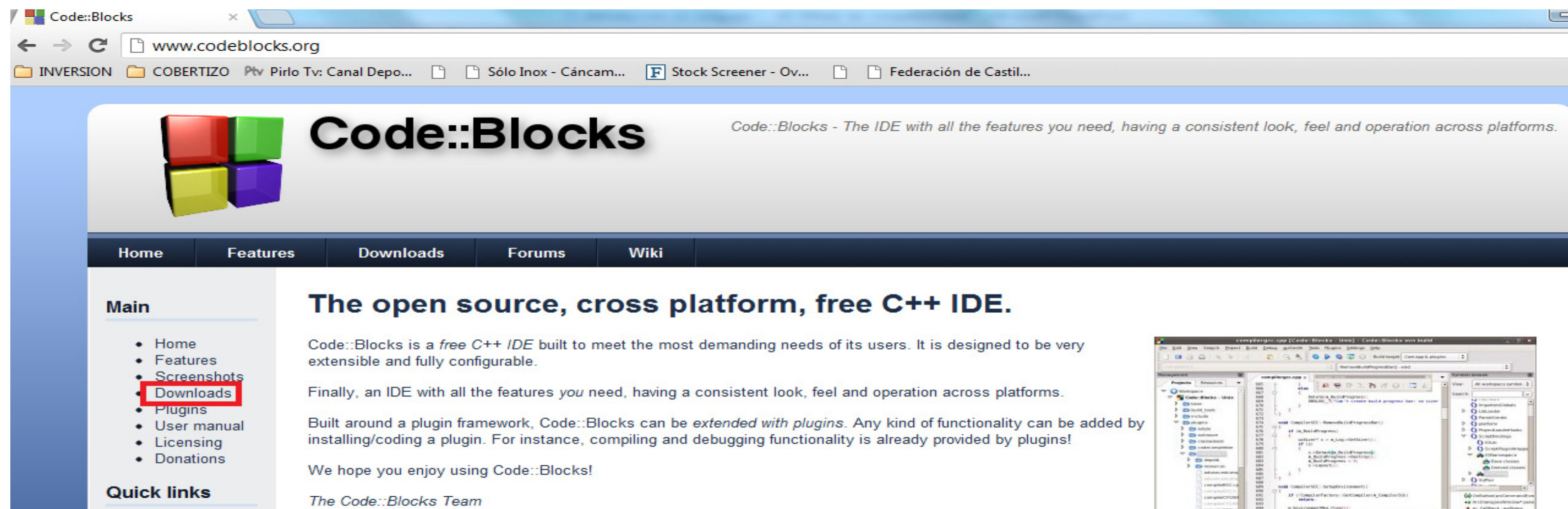
Programa ejecutable (*.exe, *.out)

```
11010101010101010101
01001010101010000101
01010100110101001 0
10100010101010110 0
```

Lectura complementaria http://www.learncpp.com/cpp-tutorial/04-introduction-to-development/

Del C al C++

CodeBlocks y primer programa C++



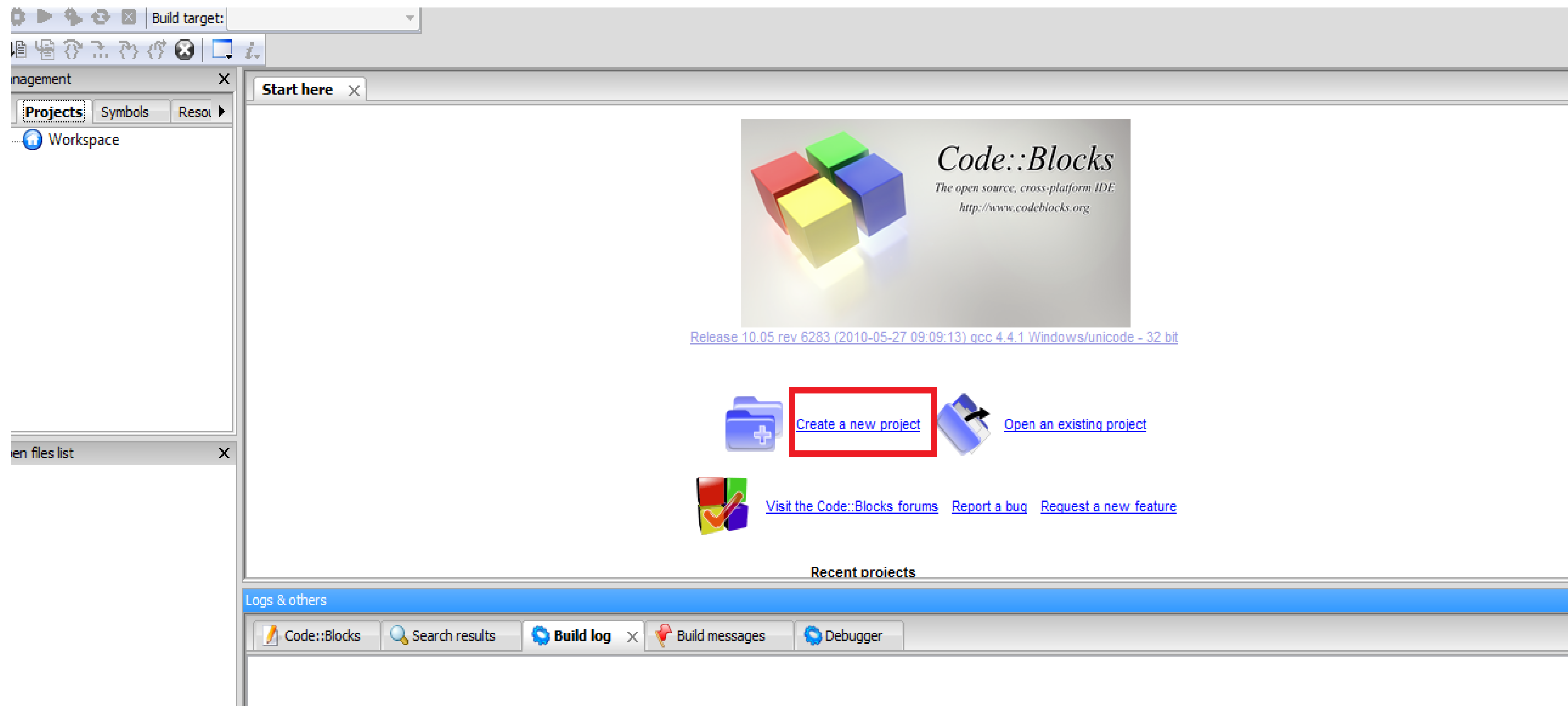
- **CodeBlocks** brinda una **IDE** (ambiente integrado de desarrollo):
  - **Editar** programa fuente
  - **Compilar**
  - **Enlazar**
  - **Depurar**
- Puede utilizar diversos compiladores (usaremos **gcc** para Windows **MinGW**)

- Más información en:
  - http://www.codeblocks.org/
  - http://es.wikipedia.org/wiki/Code::Blocks

Del C al C++

CodeBlocks y primer programa C++

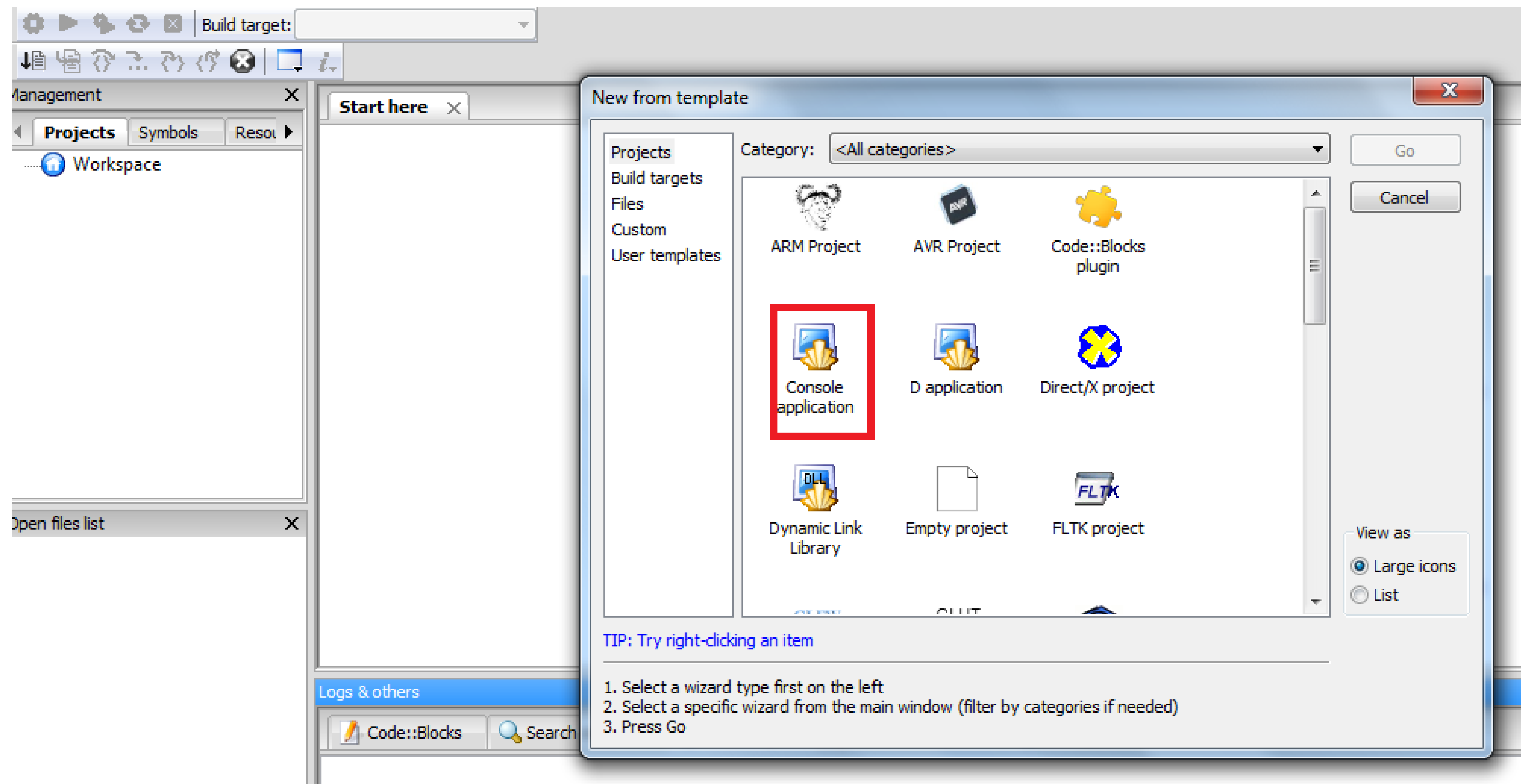
- Para programas de cierta complejidad y donde se quiera acceder a las faciliades de depuración (debugger) se recomienda **crear proyecto**.



Del C al C++

CodeBlocks y primer programa C++

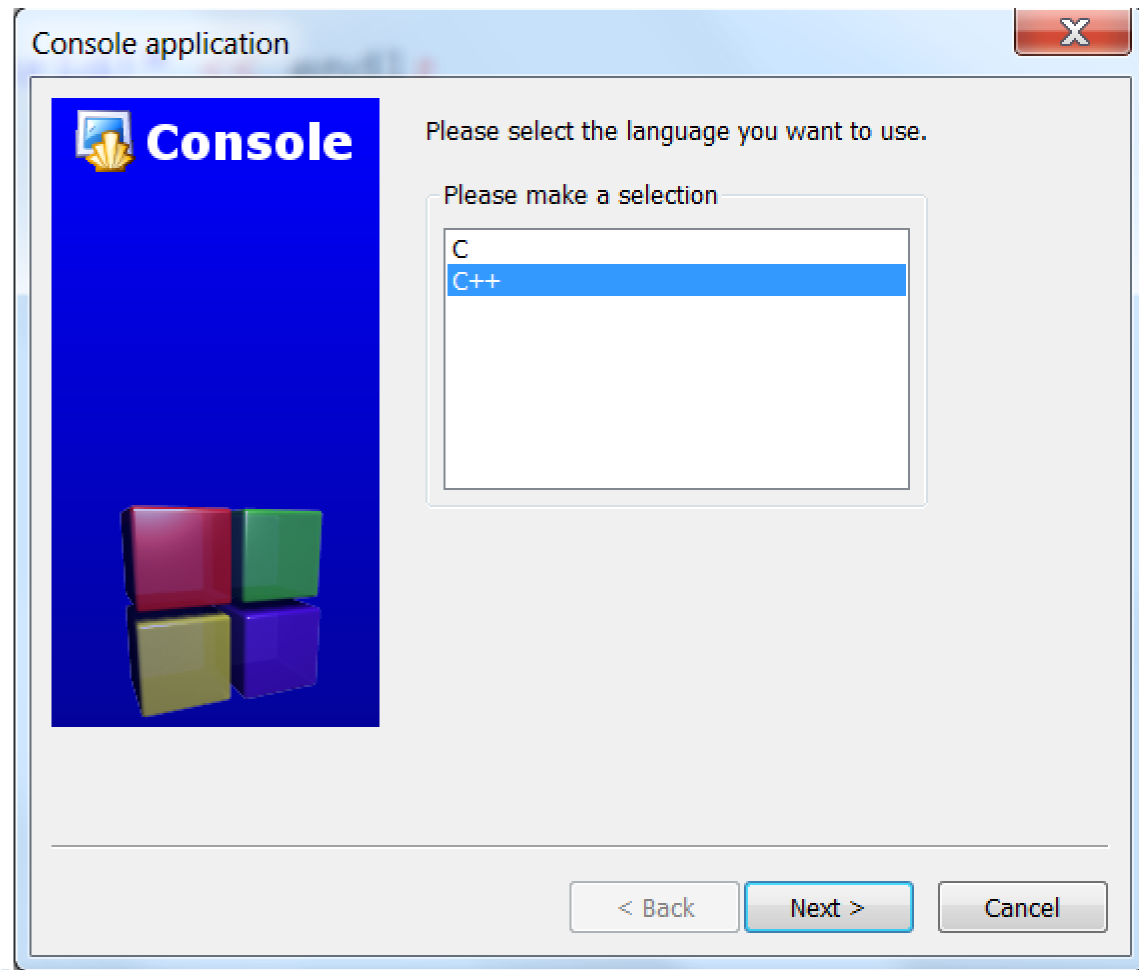
- Crearemos aplicaciones de **consola**



Del C al C++

CodeBlocks y primer programa C++

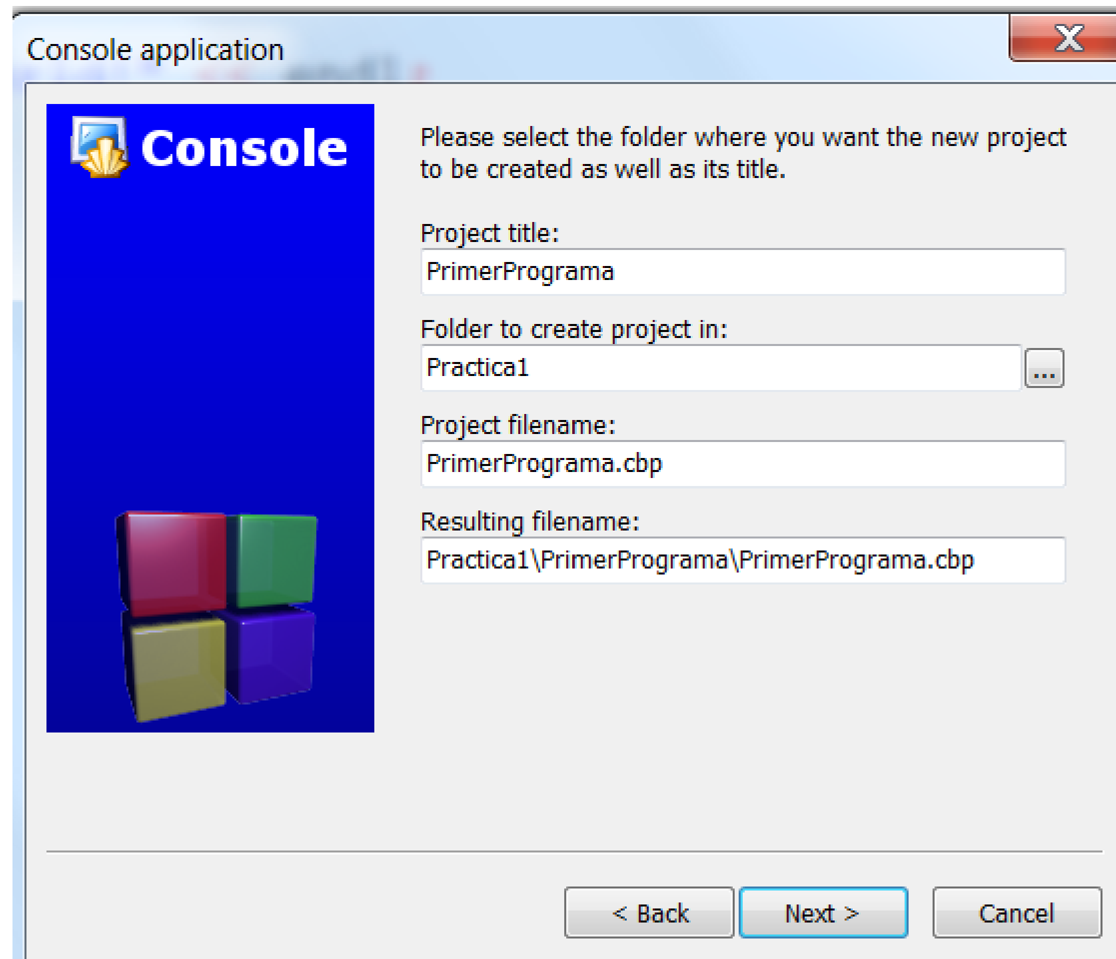
- Crearemos aplicaciones de **consola** y programaremos en C++



Del C al C++

CodeBlocks y primer programa C++

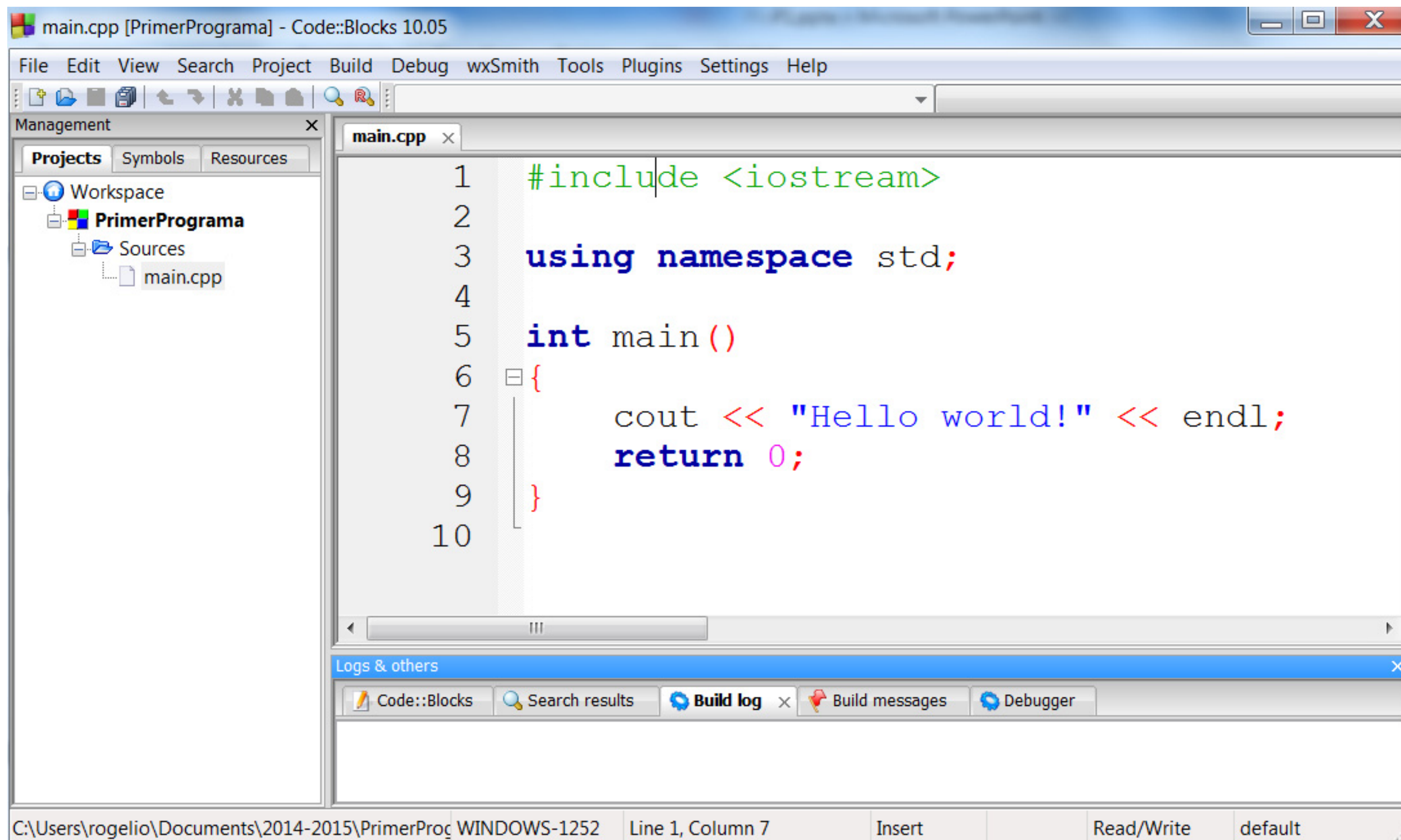
- Dar nombre de proyecto y carpeta de trabajo



Del C al C++

CodeBlocks y primer programa C++

- ¿Qué hace? ¿Cómo lo haríamos en C? Comparar



Del C al C++

CodeBlocks y primer programa C++

Seleccionar :
Settings->Compiler

Asegurar que se elige versión v11 del compilador

Del C al C++

CodeBlocks y primer programa C++

- ¿Qué hace? ¿Cómo lo haríamos en C? Comparar

Extensión *.cpp

No hay que poner extensión a los ficheros de cabecera de la biblioteca estándar de C++

main.cpp [PrimerPrograma] - Code::Blocks 10.05

File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Help

Management: Projects | Symbols | Resources

Workspace
PrimerPrograma
Sources
main.cpp

main.cpp

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Logs & others: Code::Blocks | Search results | Build log | Build messages | Debugger

C:\Users\rogelio\Documents\2014-2015\PrimerProg WINDOWS-1252 Line 1, Column 7 Insert Read/Write default

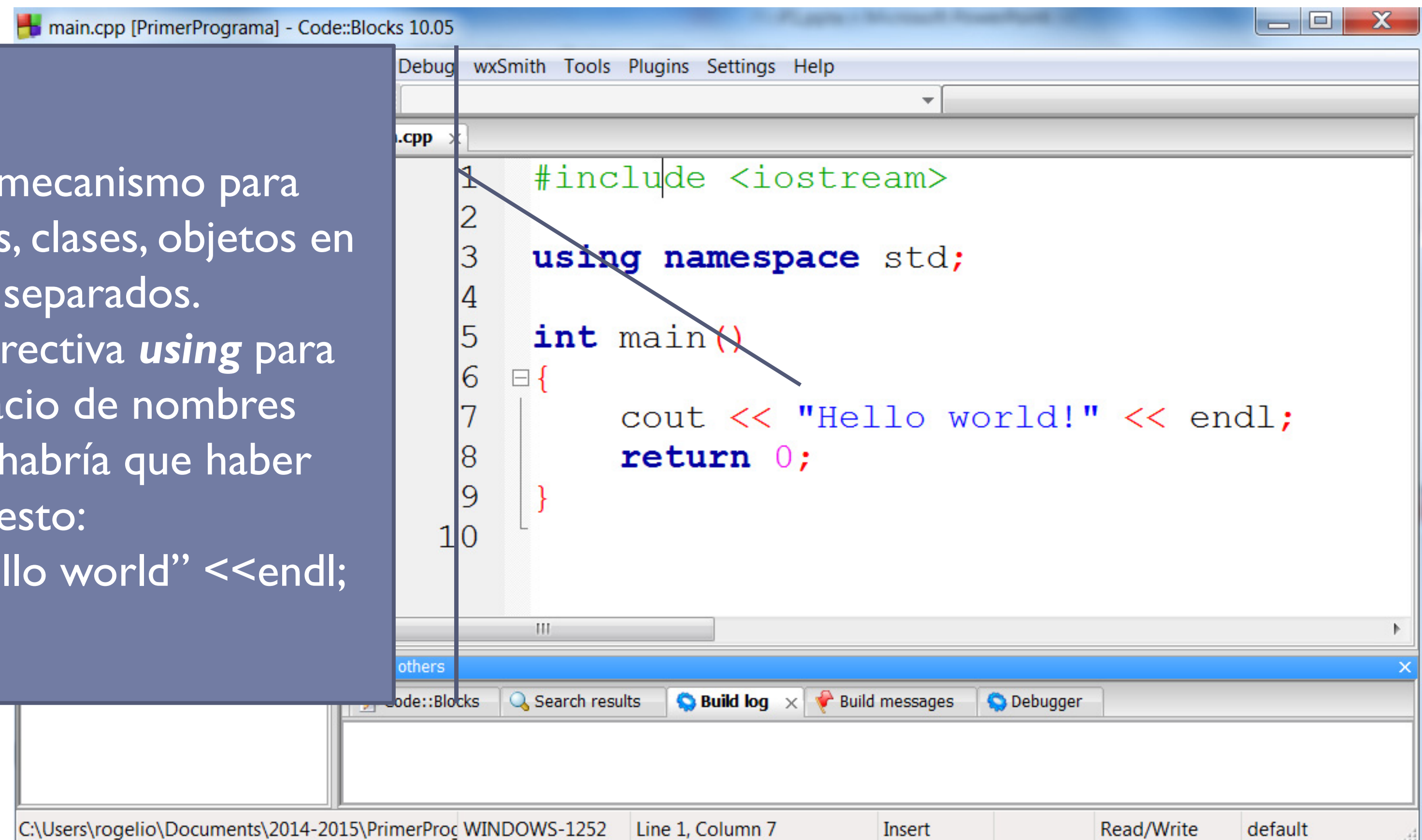
- http://www.learncpp.com/cpp-tutorial/11-structure-of-a-program/

Del C al C++

CodeBlocks y primer programa C++

- ¿Qué hace? ¿Cómo lo haríamos en C? Comparar

namespace: mecanismo para agrupar funciones, clases, objetos en espacios separados.
Si no se pone directiva ***using*** para activar el espacio de nombres estándar (std) habría que haber puesto:
std::cout << “Hello world” <<endl;

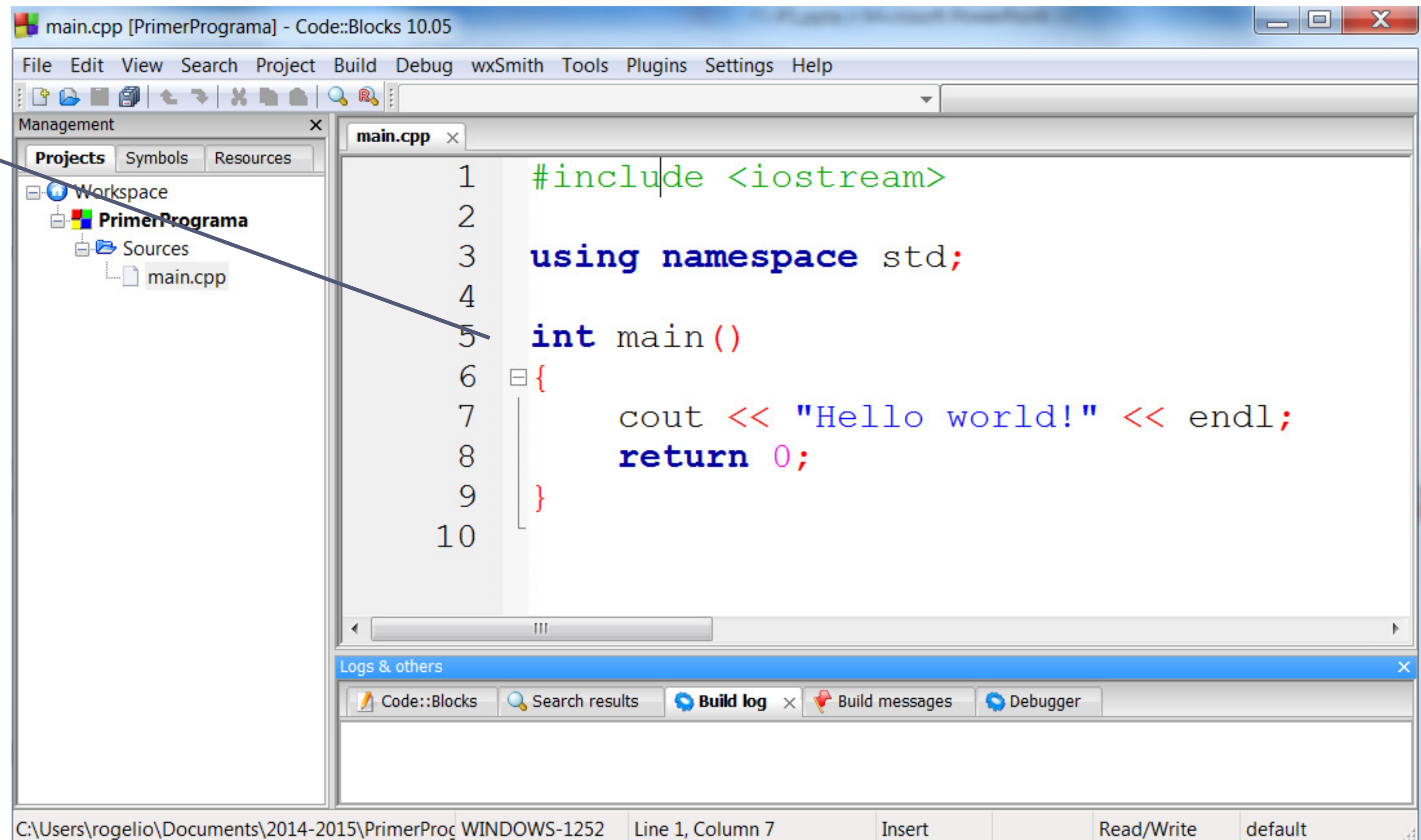


Del C al C++

CodeBlocks y primer programa C++

- ¿Qué hace? ¿Cómo lo haríamos en C? Comparar

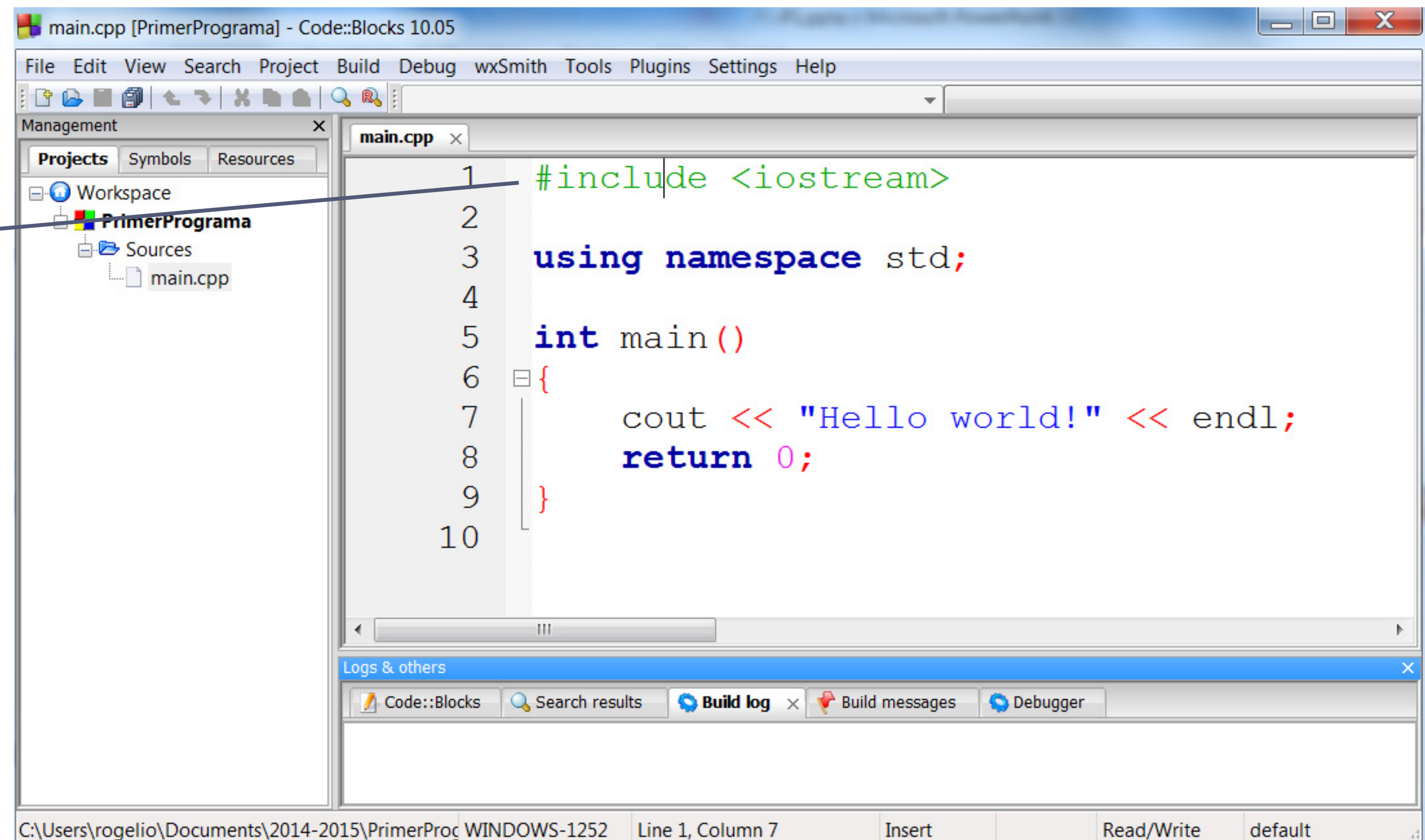
Todo programa tiene una función **main**



Del C al C++

CodeBlocks y primer programa C++

- ¿Qué hace? ¿Cómo lo haríamos en C? Comparar



- Se deben incluir ficheros de cabecera.
- Si es de la librería estándar no lleva extensión

Del C al C++

CodeBlocks y primer programa C++

- ¿Qué hace? ¿Cómo lo haríamos en C? Comparar

- Los **identificadores** (nombre de clases, objetos, funciones) se organizan es **espacios de nombres**
- Con **using** incluimos por defecto el llamado **std** (estándar)

main.cpp [PrimerPrograma] - Code::Blocks 10.05

File Edit View Search Project Build Debug wxSmith Tools Plugins Settings Help

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}

```

Logs & others: Code::Blocks | Search results | Build log | Build messages | Debugger

C:\Users\rogelio\Documents\2014-2015\PrimerProg WINDOWS-1252 Line 1, Column 7 Insert Read/Write default

Del C al C++

CodeBlocks y primer programa C++

- ¿Qué hace? ¿Cómo lo haríamos en C? Comparar

- La salida por la consola se realiza mediante el objeto predefinido **cout**
- Se redefine el operador **<<** de manera que la cadena de caracteres a su derecha se envía a **cout** para salida por pantalla
- **endl** representa cambio de línea

main.cpp [PrimerPrograma] - Code::Blocks 10.05

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello world!" << endl;
8      return 0;
9  }
10
```

Del C al C++ E/S básica en C++ mediante consola

```
#include <iostream>

using namespace std;

int main()
{
    char nombre[30];
    int edad;
    double altura;

    cout << "Diga su mombre" << endl;
    cin >> nombre;
    cout << "Hola " << nombre << " Dime edad y altura (m) " << endl;
    cin >> edad >> altura;
    cout << nombre << " tiene " << edad <<" annos y " << altura << " metros de altura." <<endl;
    return 0;
}
```

- **Ej:** Compilar y ejecutar programa anterior. ¿Se puede introducir nombre y apellidos? ¿Qué pasa cuándo lo intentas?

E/S básica en C++ mediante consola

```
#include <iostream>

using namespace std;

int main()
{
    //char nombre[30];
    string nombre;
    int edad;
    double altura;

    cout << "Diga su mombre" << endl;
    //cin >> nombre;
    getline(cin,nombre);
    cout << "Hola " << nombre << " Dime edad y altura (m) " << endl;
    cin >> edad >> altura;
    cout << nombre << " tiene " << edad <<" annos y " << altura << " metros de altura." <<endl;
    return 0;
}
```

Utilizamos un tipo de datos nuevo (una clase) para las cadenas de caracteres. Observe que no hay que poner el tamaño. Veremos la clase string detalladamente más adelante.

Función **getline** para leer de la consola una cadena de caracteres que contenga espacios.

Informática Industrial

Práctica 2. Nuevos tipos de datos. Paso por referencia.
Sobrecarga de funciones.

Del C al C++

- Tipos de datos. Tipo de dato **bool.** Datos enumerados.
- Elementos básicos de E/S con la consola
- Funciones en C++
  - Declaración, definición de funciones
  - Alcance y tiempo de vida de las variables. Una aplicación C++ en la memoria del ordenador
  - Proceso de llamada a una función durante la ejecución
  - Paso de parámetros por valor y por referencia.
  - Sobrecarga de funciones
  - Parámetros por defecto

Del C al C++

- Tipos de datos básicos

Grupo	nombre	Nota
lógica	***bool***	Puede representar sólo dos estados: *true* y *false*
caracter*	***char***	Un byte.
enteros*	***int***	No menor ***char***
	long	No menor ***int***
	long long	No menor que ***long***
reales	***float***	
	double	Mayor precisión que **float**
	long double	Mayor precisión que ***double***

*Hay versiones **signed** y **unsigned.** Ej:
signed char va de -128 a 127
unsigned char va de 0 a 255

Del C al C++

Serán utilizados con preferencia durante el curso

- Tipos de datos básicos

Grupo	nombre	Nota
lógica	***bool***	Puede representar sólo dos estados: *true* y *false*
caracter*	***char***	Un byte.
enteros*	***int***	No menor ***char***
	long	No menor ***int***
	long long	No menor que ***long***
reales	***float***	
	double	Mayor precisión que **float**
	long double	Mayor precisión que ***double***

*Hay versiones **signed** y **unsigned.** Ej:

signed char va de -128 a 127

unsigned char va de 0 a 255

Del C al C++

- Ejemplo del uso del nuevo tipo de datos ***bool***.

```
#include <iostream>

using namespace std;

int main()
{
    bool cierto;  //Se define cierto como de tipo bool

    //hay dos nuevas cosntante literales false y true
    cierto= false && true;
    cout << "valor de cierto " << cierto <<  endl;
    cierto= true || cierto;
    cout << "valor de cierto " << cierto <<  endl;

    //Sigue ocurriendo que un valor de cero equivale a false y un valor
    //diferente de cero es equivalente a true
    cierto=3;
    cierto = !cierto;

    if (!cierto)
        cout << "cierto es false (o sea cero) |" << cierto << endl;


    return 0;
}
```

Del C al C++

Tipos enumerados

C++ permiten definir **tipo nuevos enumerando los elementos** que este nuevo tipo puede contener. El uso de los tipo de datos enumerados mejora la legibilidad del código.

enum Meses {enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre, noviembre, diciembre};

En lo anterior y por defecto, se asigna un entero, comenzando por cero, a cada dato enumerado del dominio. Esta asignación se puede alterar a voluntad:

enum Notas {Matricula=10, Sobresaliente = 9, notable = 7, aprobado = 5, suspenso = 0};

Ejemplo enumerados

```
#include <iostream>
using namespace std;

enum CalidadDato {mala, incierta, buena};
CalidadDato ObtieneCalidad();

int main()
{
    CalidadDato sensor;

    sensor = ObtieneCalidad();
    switch (sensor)
    {
        case mala:
                cout << "mal dato" << endl;
                break;
        case incierta:
                cout << "no se sabe" << endl;
                break;
        case buena:
                cout << "sensor funciona correctamente" << endl;
                break;
    }
    return 0;
}
CalidadDato ObtieneCalidad()
{
    return buena;
}
```

Ejemplo enumerados

```
#include <iostream>
using namespace std;

enum CalidadDato {mala, incierta, buena};
CalidadDato ObtieneCalidad();

int main()
{
    CalidadDato sensor;

    sensor = ObtieneCalidad();
    switch (sensor)
    {
        case mala:
                cout << "mal dato" << endl;
                break;
        case incierta:
                cout << "no se sabe" << endl;
                break;
        case buena:
                cout << "sensor funciona correctamente" << endl;
                break;
    }
    return 0;
}
CalidadDato ObtieneCalidad()
{
    return buena;
}
```

¿Que saldría por pantalla?

return (CalidadDato) 2;

Del C al C++

Alcance y tiempo de vida de las variables.

Estructura de un programa en memoria

Alcance y tiempo de vida de las variables

```
#include <iostream>
using namespace std;

int glob=1;
void primeraFuncion(int, int);
void segundaFuncion(int);
int main()
{
    int inferior=1, superior=6;

    primeraFuncion(inferior,superior);
    cout << "Global desde main " << glob << endl;
    return 0;
}
void primeraFuncion(int inf, int sup)
{
    int i;

    for (i=inf; i < sup;i++)
        segundaFuncion(i);
}
void segundaFuncion(int ent)
{
    int i=0;
    cout << "Parametro: " << ent << " local i " << i ;
    cout << " global: " << glob << endl;
    glob=++glob % 2;
    i++;
}
```

Alcance y tiempo de vida de las variables

```
#include <iostream>
using namespace std;

int glob=1;
void primeraFuncion(int, int);
void segundaFuncion(int);
int main()
{
    int inferior=1, superior=6;

    primeraFuncion(inferior,superior);
    cout << "Global desde main " << glob << endl;
    return 0;
}
void primeraFuncion(int inf, int sup)
{
    int i;

    for (i=inf; i < sup;i++)
        segundaFuncion(i);
}
void segundaFuncion(int ent)
{
    int i=0;
    cout << "Parametro: " << ent << " local i " << i ;
    cout << " global: " << glob << endl;
    glob=++glob % 2;
    i++;
}
```

Variable definida fuera de toda función:
Alcance (*scope*): **global** (pueden ser accedidas desde cualquier función)
Tiempo de vida (*lifetime*): mantien el valor **durante toda la ejecución del programa**

Parámetros y variables locales definidas dentro de la funciones
Alcance (*scope*): **local** (pueden ser accedidas desde la función en las que son definidas)
Tiempo de vida (*lifetime*): mantienen su valor mientras la ejecución del programa se encuentra en la función donde están definidas.

Estructura de un programa en memoria

Direcciones bajas de memoria

Servicios del S.O.

Núcleo S.O (kernel)

Instrucciones

Datos globales

Variables con **tiempo de vida ilimitado** (típicamente variables de **alcance global**)

Proceso (programa del usuario, cargado en memoria y ejecutándose)

Heap (montículo)

Memoria asignada **dinámicamente** (pedida por el proceso al S.O.)

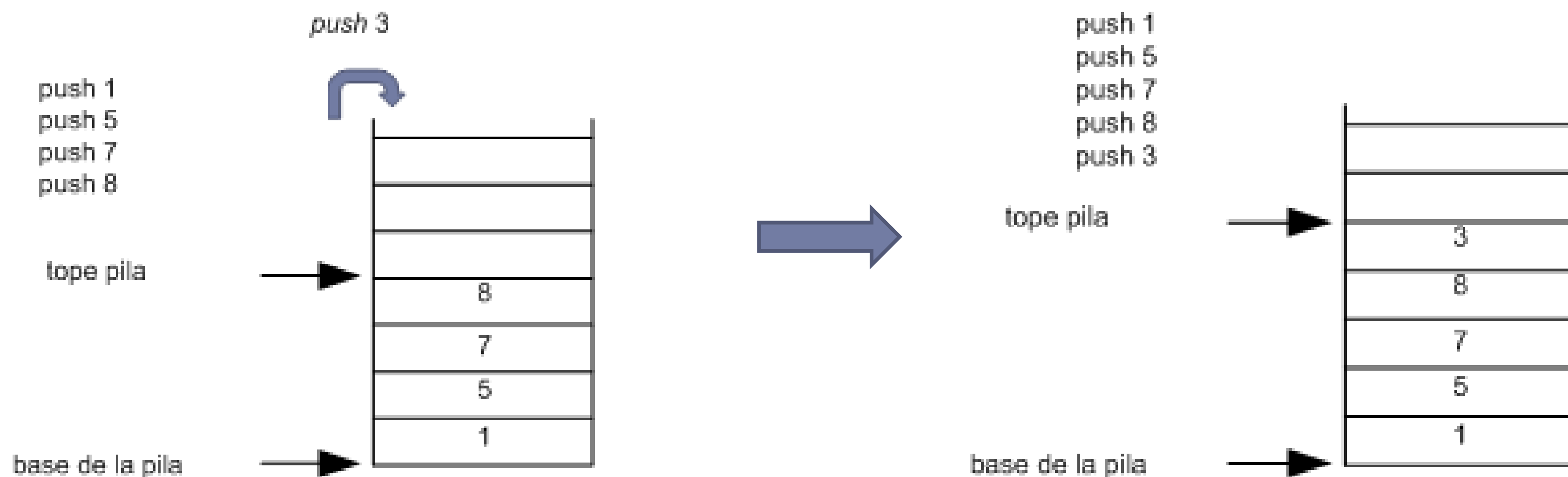
Stack (*pila*)

Variables automáticas, de **alcance local** a la función donde se definen y **tiempo de vida limitado** al momento en que la función está ejecutándose

Direcciones altas de memoria

Funcionamiento de una pila o *stack*

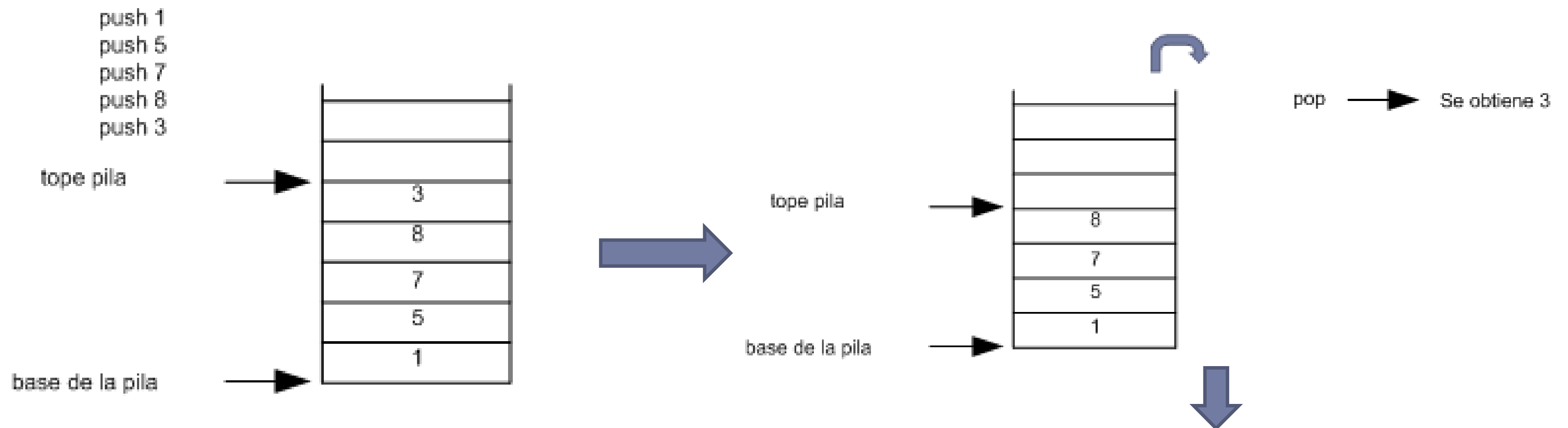
Una pila es una colección de datos en la que tanto las **inserciones** (operación *push*) como las **eliminaciones** (operación *pop*) de nuevos elementos tienen lugar por el mismo extremo llamado tope o cabeza de la pila. Se les suele denominar como de tipo LIFO (*last in first out*).



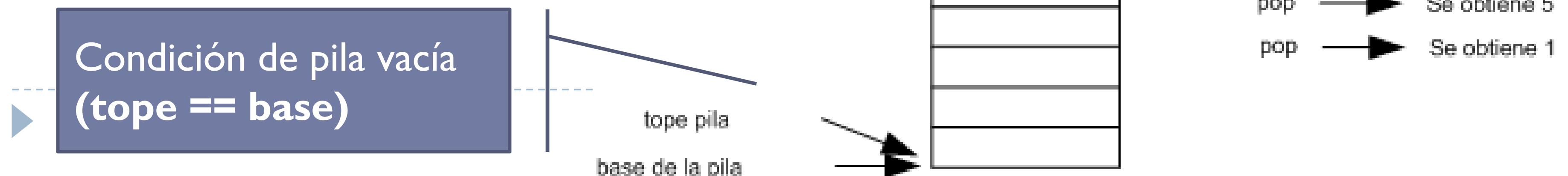
Operación de inserción en la pila

Funcionamiento de una pila o *stack*

Una pila es una colección de datos en la que tanto las **inserciones** (operación *push*) como las **eliminaciones** (operación *pop*) de nuevos elementos tienen lugar por el mismo extremo llamado tope o cabeza de la pila. Se les suele denominar como de tipo LIFO (*last in first out*).



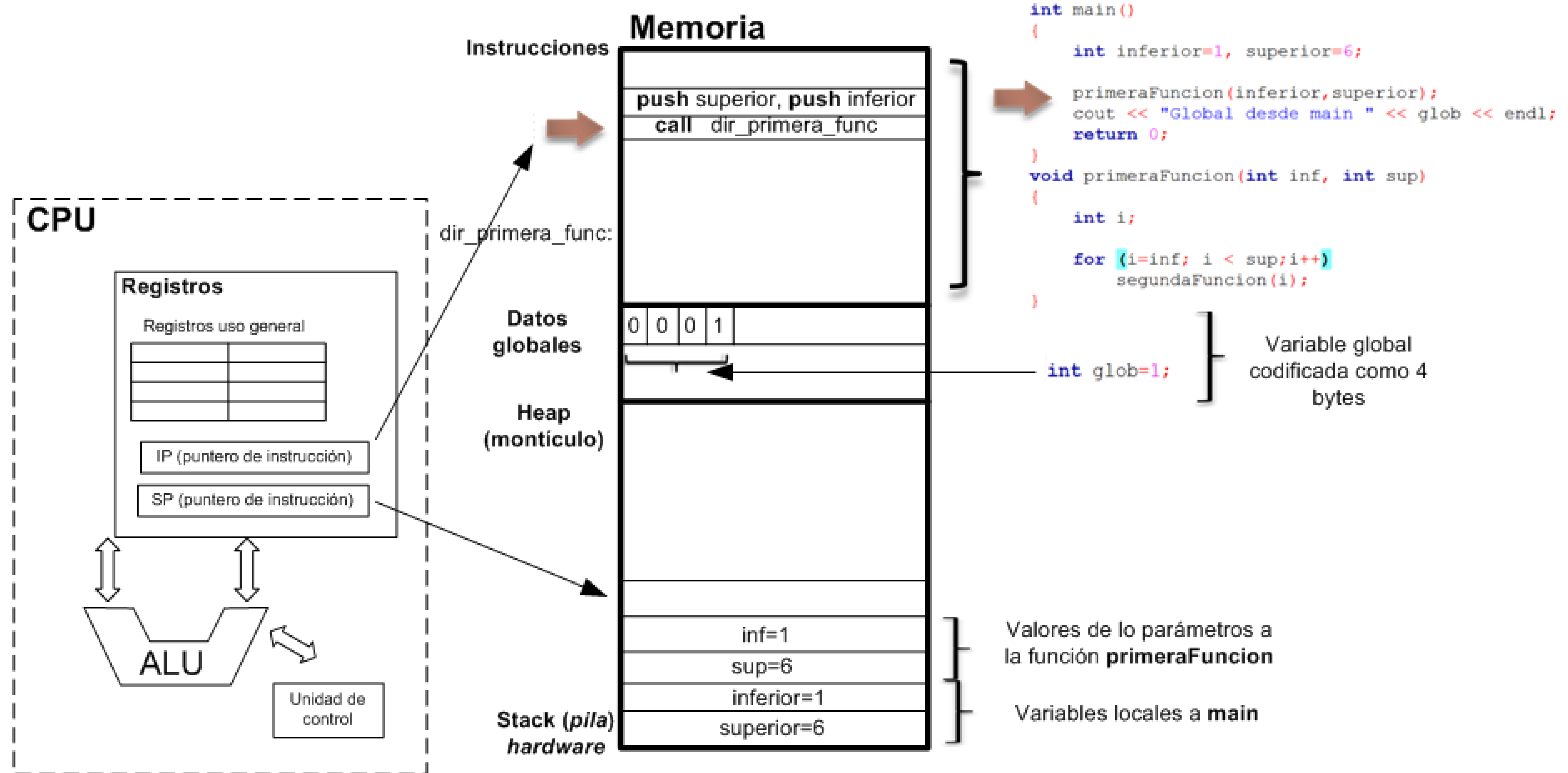
Operación de eliminación de la pila



Condición de pila vacía (tope == base)

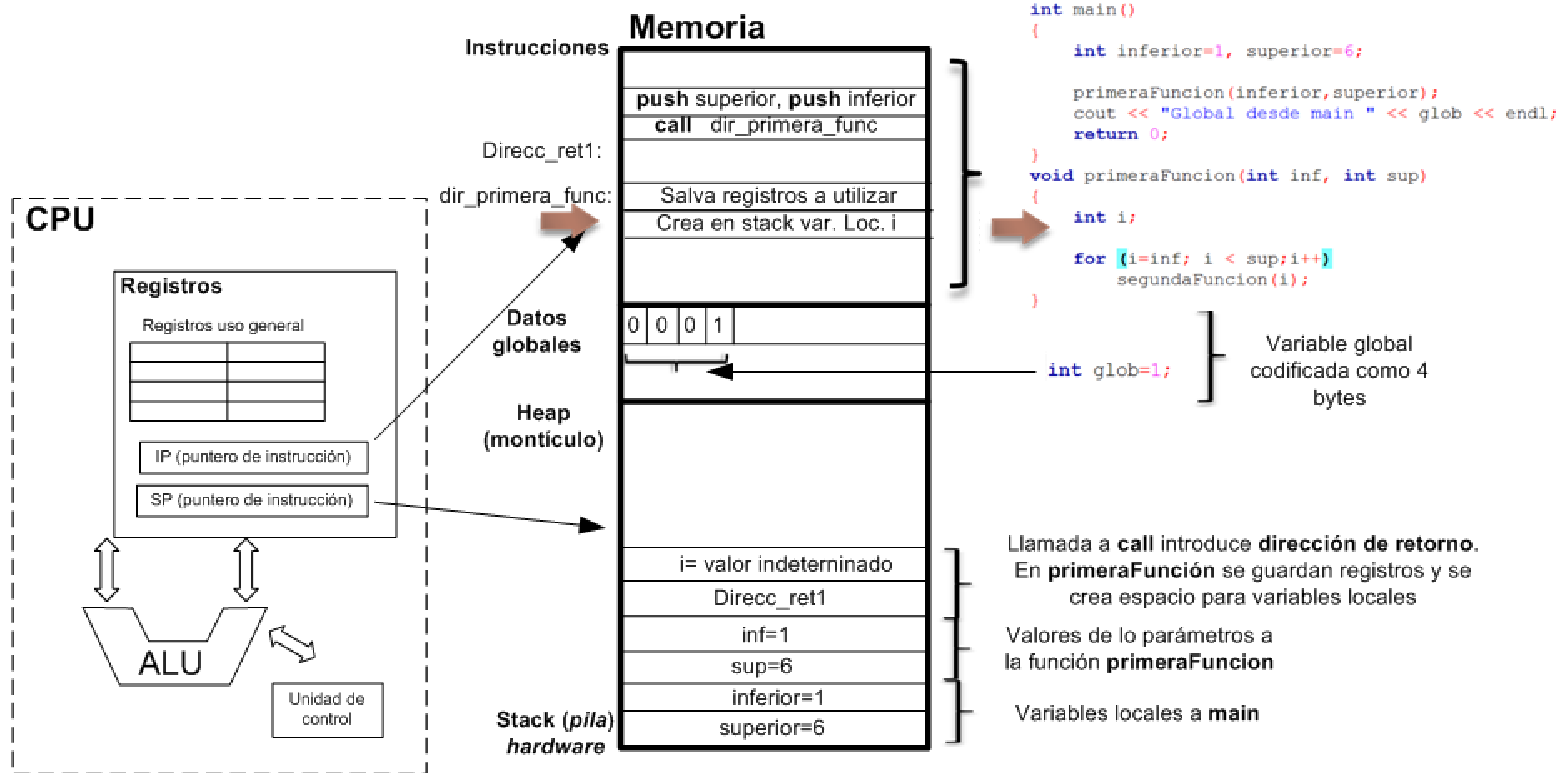
Llamada a función y creación de variables locales automáticas

- Las variable locales (incluidas las de main) se crean en el stack
- Antes de llamar a una función se introducen los valores reales de los parámetros en el stack (convención usual: de derecha a izquierda)



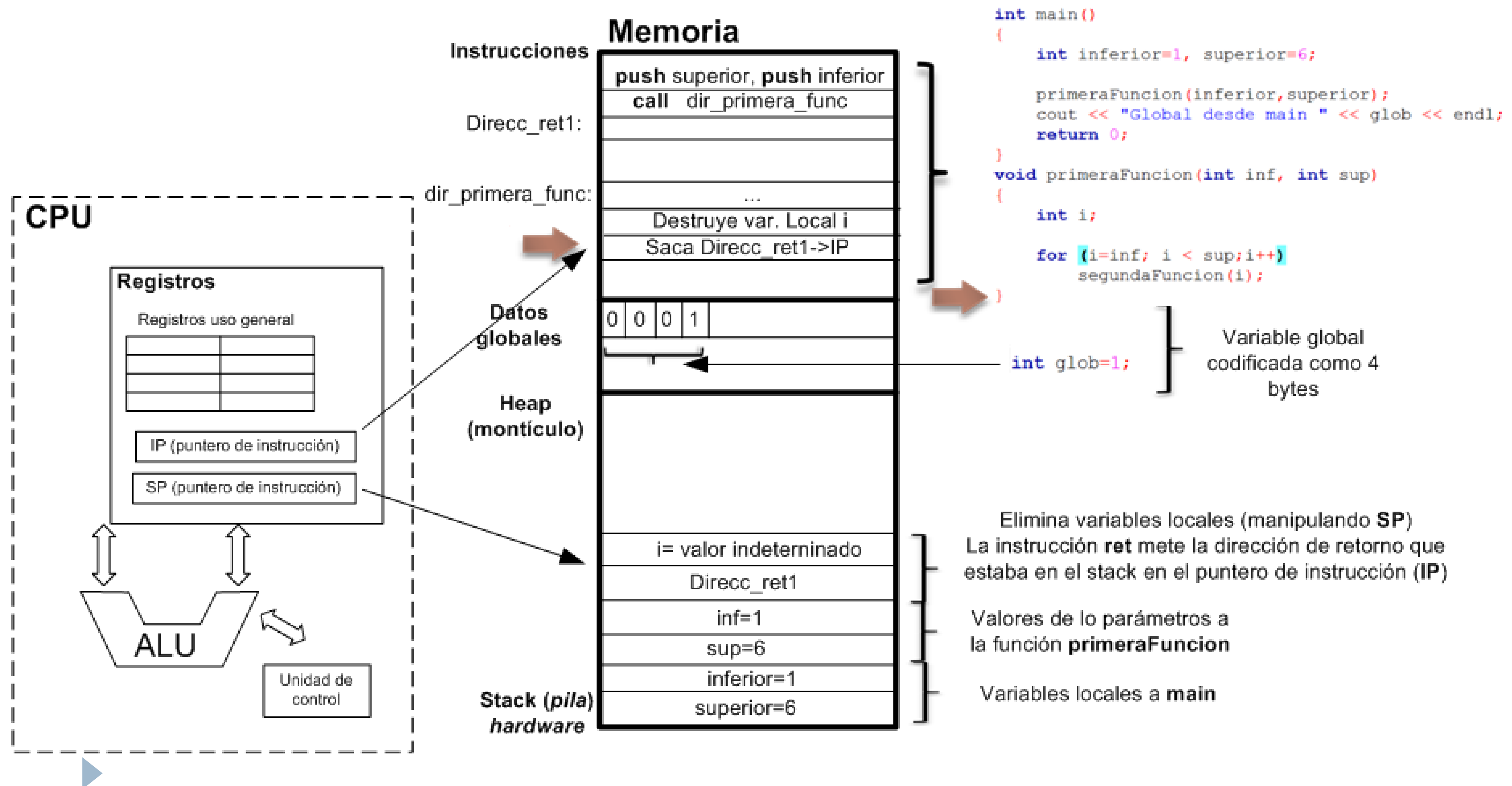
Llamada a función y creación de variables locales automáticas

- Llamada a **call** introduce automáticamente en el stack la dirección de la instrucción siguiente a la que ha producido el **call**.
- Ya en la nueva función, se guardan en el stack los registos que se vayan a modificar, y se crea espacio en el stack para las variables locales.



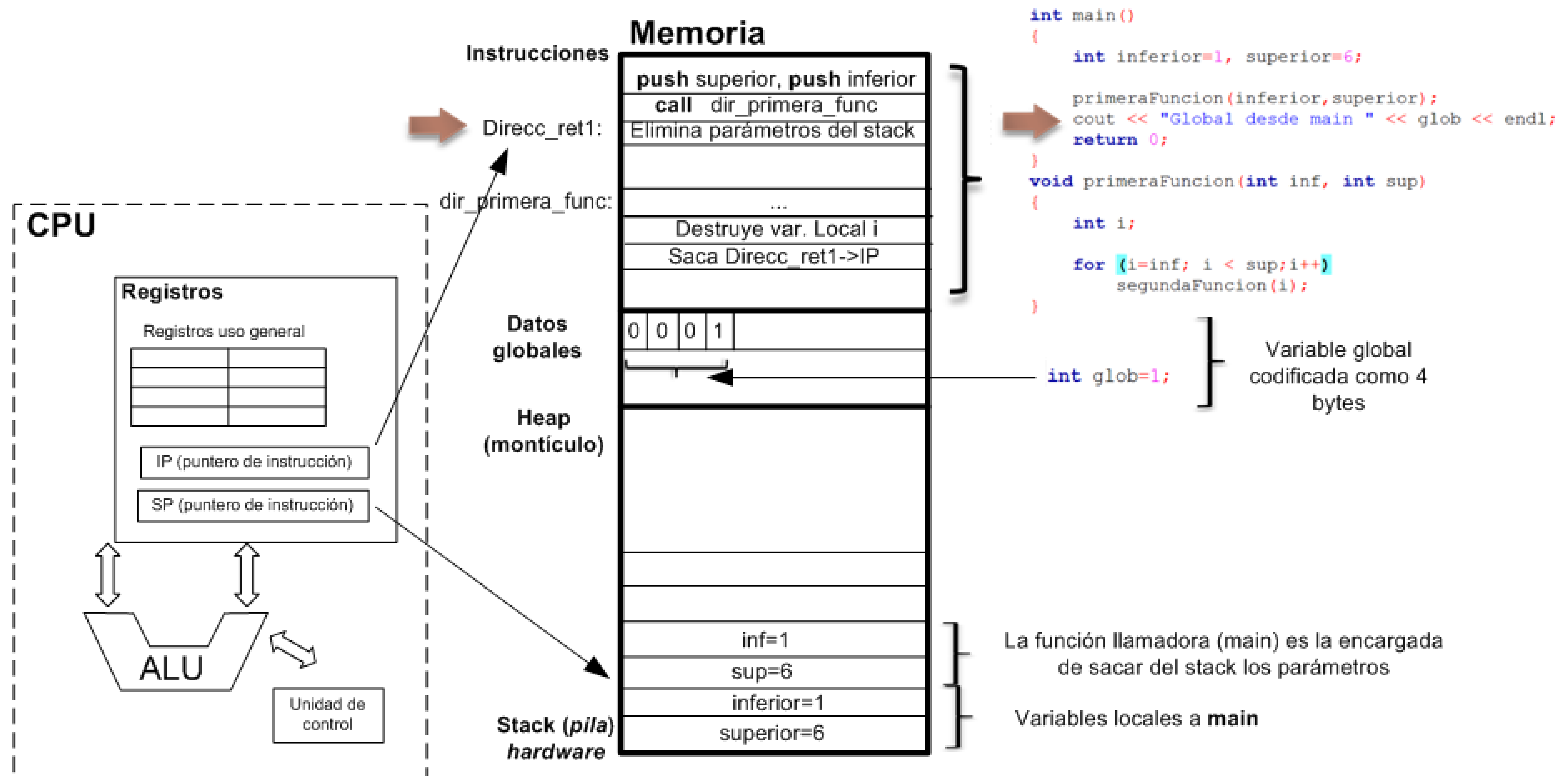
Llamada a función y creación de variables locales automáticas

- Al terminar, la función llamada (**primeraFuncion**) elimina las variables locales del stack.
- Ejecuta instrucción **ret** que automáticamente obtiene del stack la dirección a la que debe retornar en la función que llama (**main**)



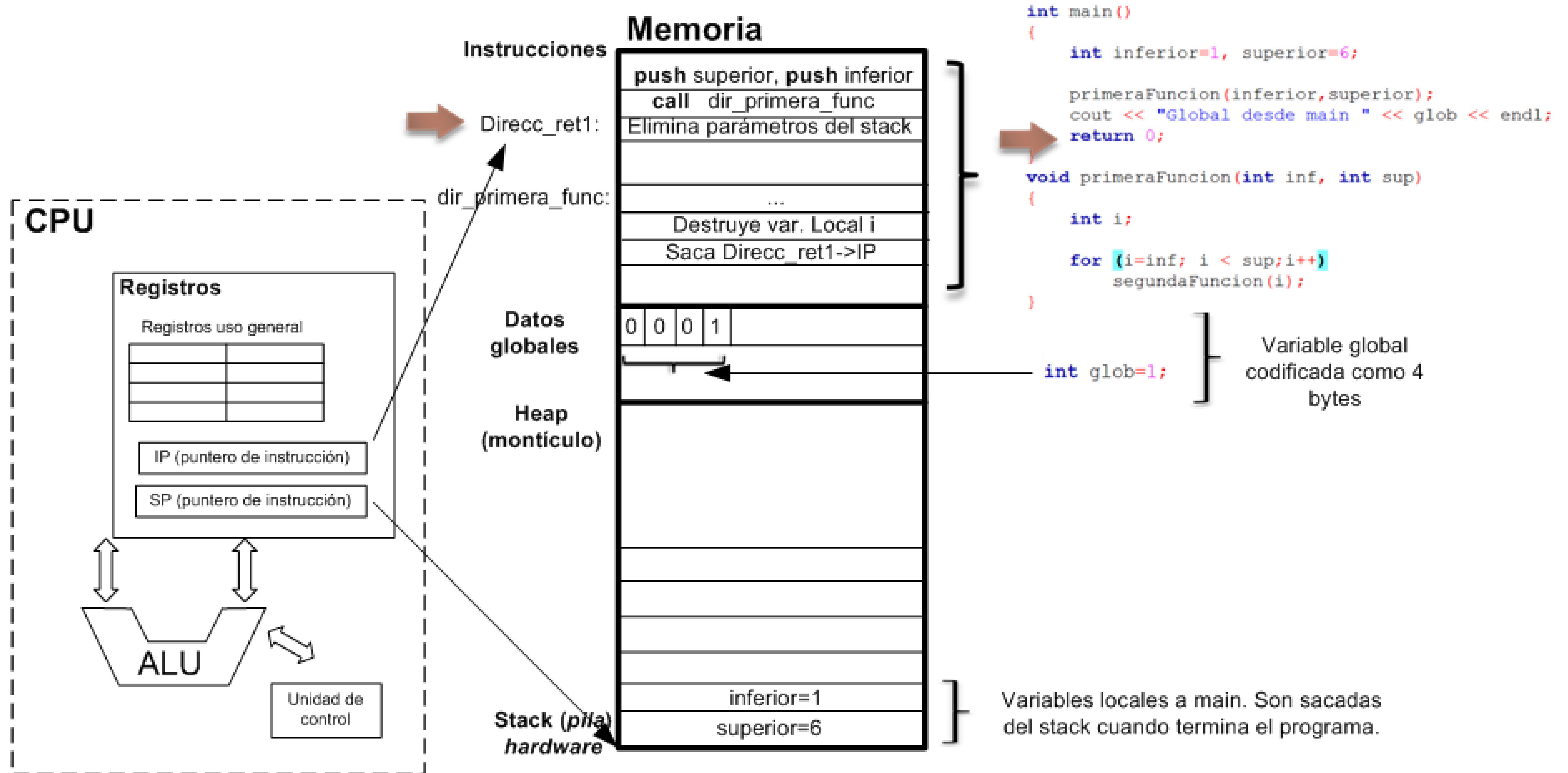
Llamada a función y creación de variables locales automáticas

- La función que llama (**main**) debe sacar los parámetros reales del stack manipulando el puntero del stack.
- En este momento sólo quedarán en el stack las variables locales de **main.**



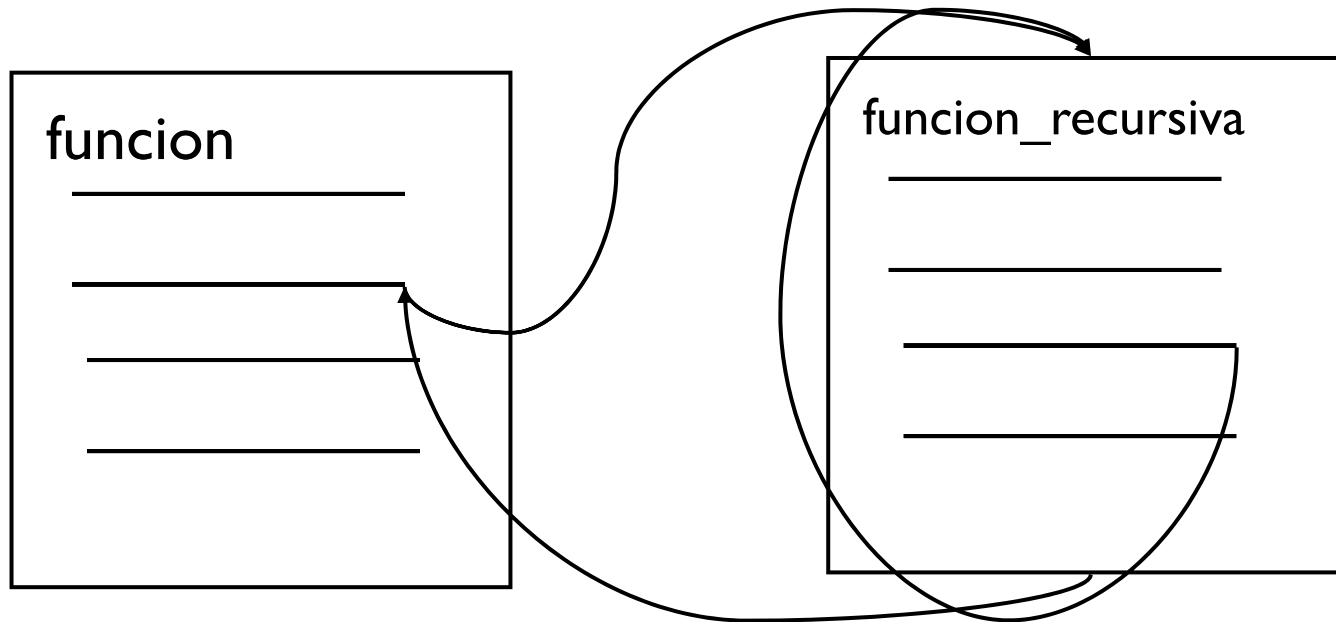
Llamada a función y creación de variables locales automáticas

- Las **variables locales a main** son eliminadas del stack cuando termina dicha función (y por tanto el programa).



Recurrencia

- El concepto de variables locales definidas en una estructura de datos de tipo **stack** permite el concepto de **recurrencia**: cuando una función se llama a sí misma repetidas veces.



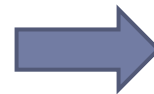
- Un ejemplo típico es su uso para el cálculo del factorial.

$$
\begin{cases} n! = 1 & \text{si } n<2 \\ n! = n*(n-1)! & \text{si } n >= 2 \end{cases}
$$

Recurrencia

```
#include <iostream>

using namespace std;
int factorial (int);
int main()
{
    int n,fact;

    cout << "Diga número para el cálculo del factorial" << endl;
    cin >> n;
    fact=factorial(n);
    cout << "El factorial de " << n << " es " << fact << endl;
    return 0;
}

int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

Comparar con definición recursiva

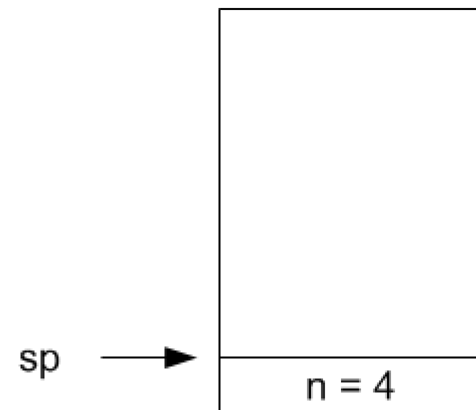


$$
\begin{cases} n! = 1 & \text{si } n<2 \\ n! = n*(n-1)! & \text{si } n >= 2 \end{cases}
$$

Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?

```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

factorial es llamado con n=4

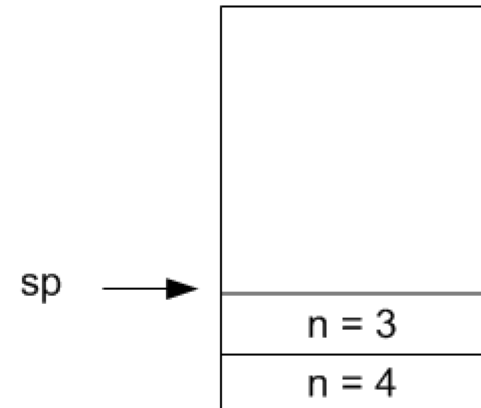


Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?

```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

factorial es llamado con n=4

Llamada recurrente con n=3



Observar que llamadas anteriores no han concluido. Por tanto, los valores anteriores (locales) de los parámetros permancen en el stack.

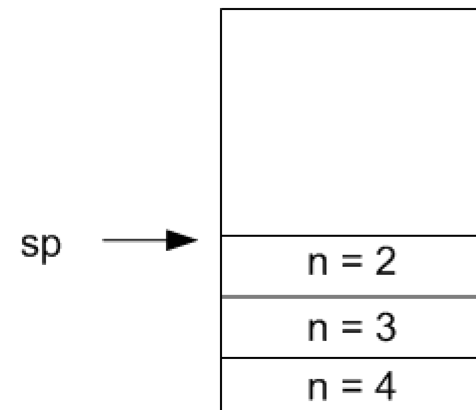
Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?

```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

factorial es llamado con n=4

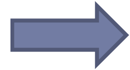
Llamada recurrente con n=3

Llamada recurrente con n=2



Observar que llamadas anteriores no han concluido. Por tanto, los valores anteriores (locales) de los parámetros permancen en el stack.

Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?



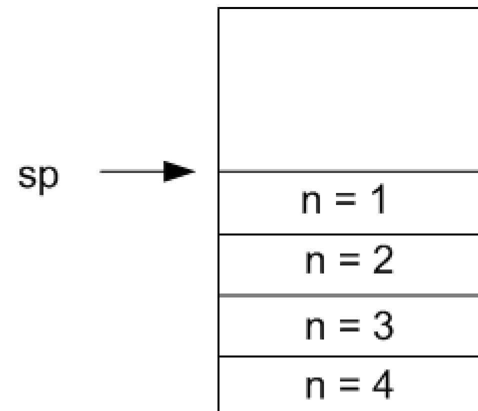
```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

factorial es llamado con n=4

Llamada recurrente con n=3

Llamada recurrente con n=2

Llamada recurrente con n=1



¿Llegados a este punto que sucede en factorial?

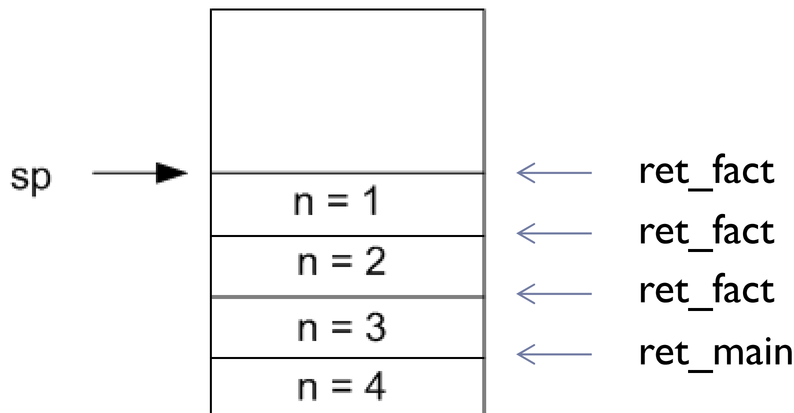
Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?

```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

ret_fact

¿Llegados a este punto que sucede en factorial?

n < 2: factorial retorna por primera vez. (devolviendo un 1) ¿A qué sentencia retorna?



ret_fact
ret_fact
ret_fact
ret_main

El *stack* mostrado es una simplificación: tambien incluye las direcciones de retorno en cada una de las llamadas a las funciones

Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?

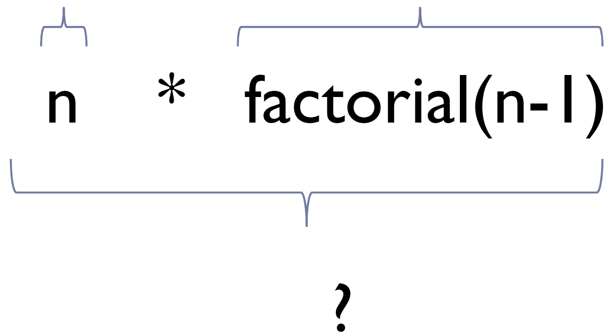
ret_fact

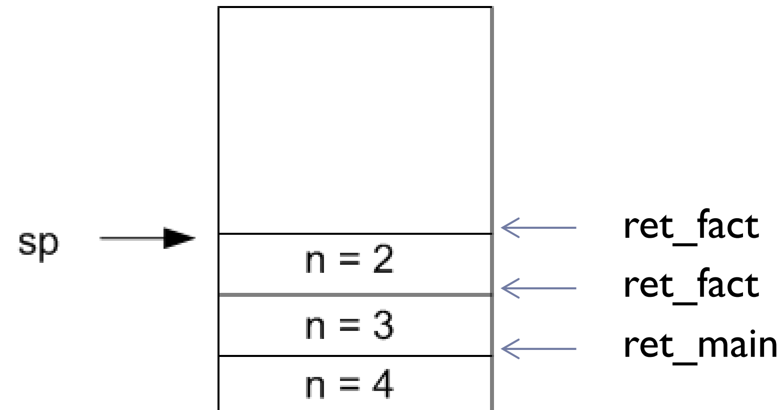
```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

Al retornar, elimina del stack la dirección de retorno y el parámetro local de la última llamada y efectúa la expresión indicada:

Valor actual de n. Ver stack

Valor que devolvió la última llamada a factorial





Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?

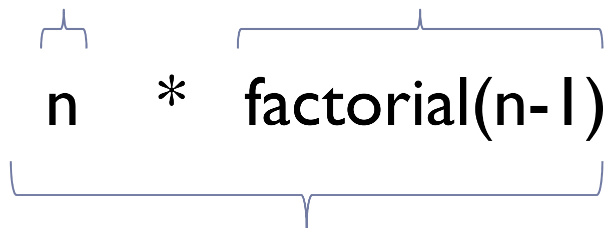
```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

ret_fact →

Al retornar, elimina del stack la dirección de retorno y el parámetro local de la última llamada y efectúa la expresión indicada:

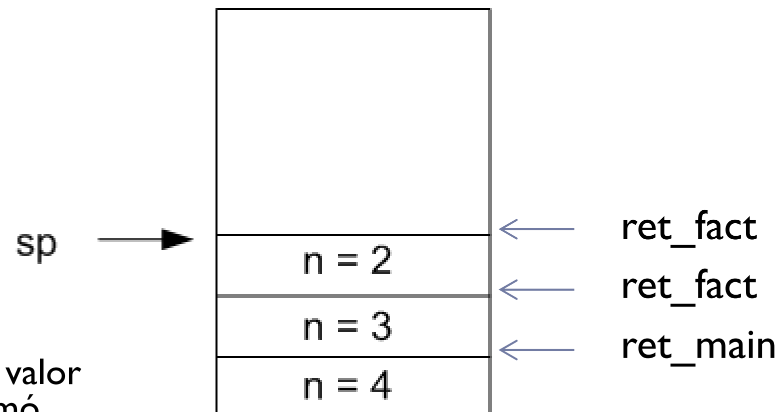
Valor actual de n. Ver stack=2

Valor que devolvió la última llamada a factorial (1)



2

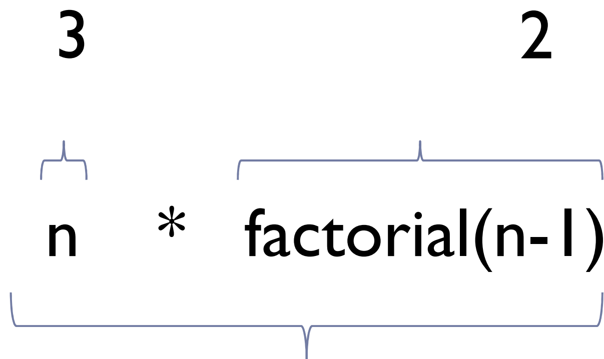
Y se devuelve este valor a la función que llamó



Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?

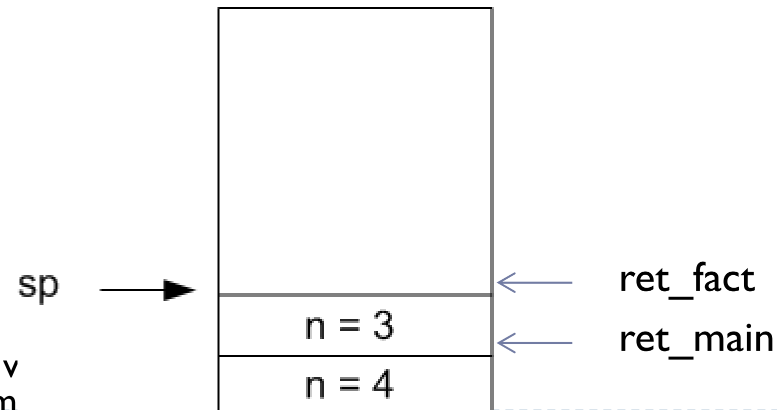
```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
ret_fact ⇒  return n*factorial(n-1);
}
```

Al retornar, elimina del stack la dirección de retorno y el parámetro local de la última llamada y efectúa la expresión indicada:



6

Y se devuleve este v
a la función que llam

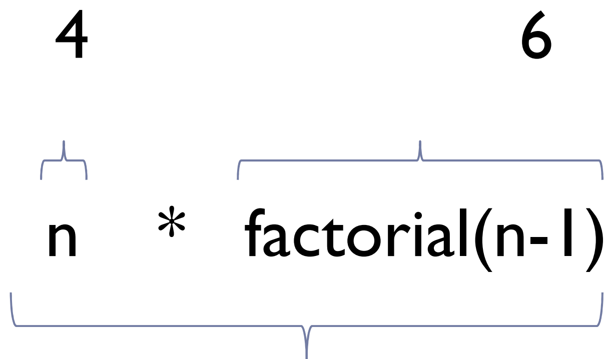


Recurrencia. ¿Por qué el definir parámetros y variables locales en el stack hace posible la recurrencia?

ret_fact

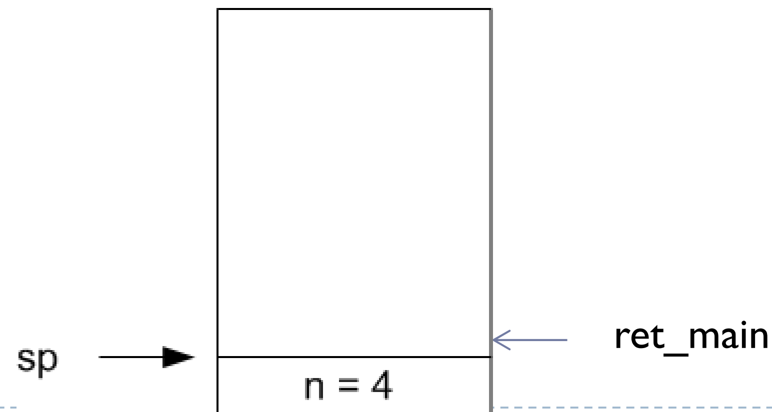
```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

Al retornar, elimina del stack la dirección de retorno y el parámetro local de la última llamada y efectúa la expresión indicada:

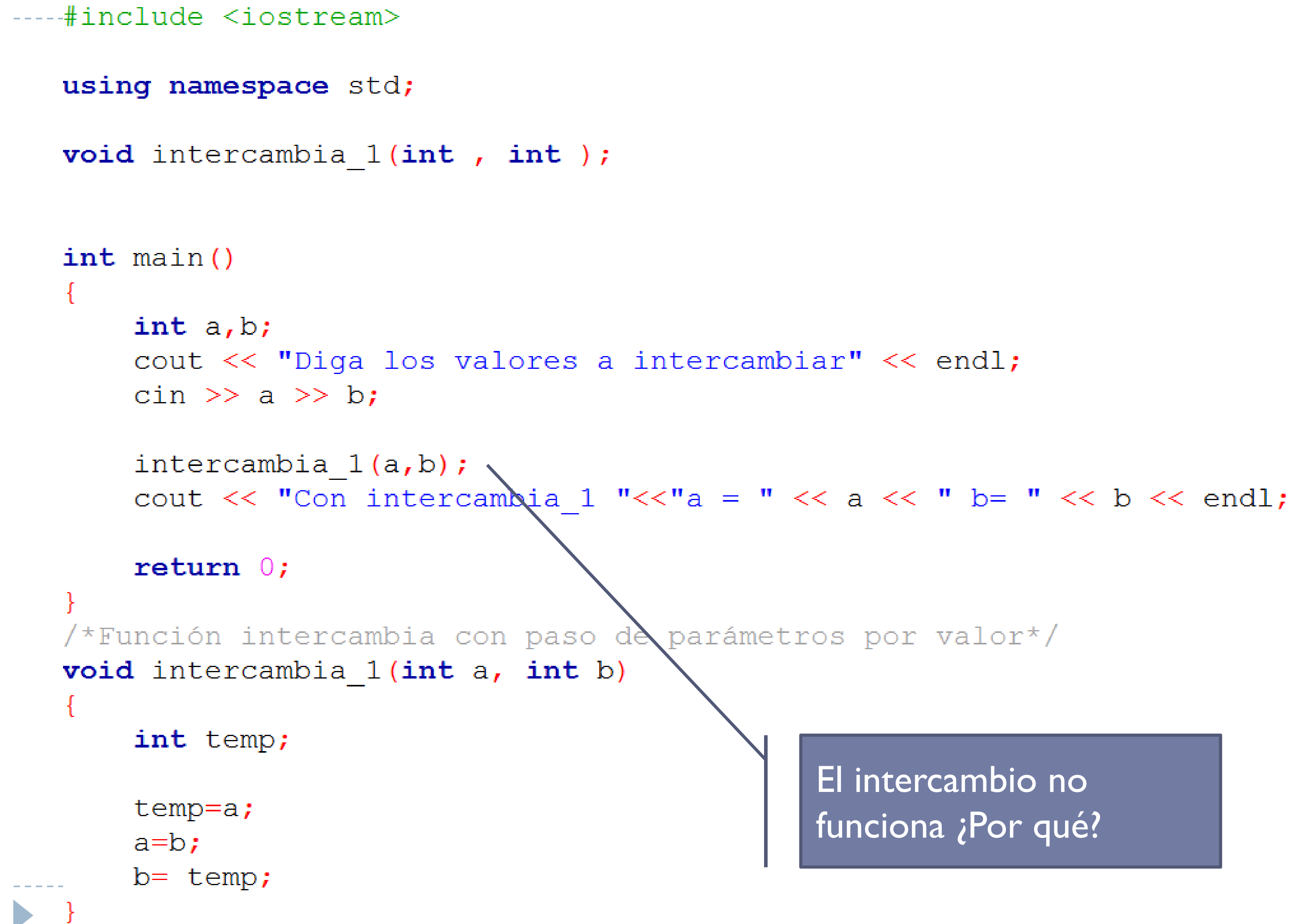


24

Y se devuelve este valor a la función main: resultado final de 4!



Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)



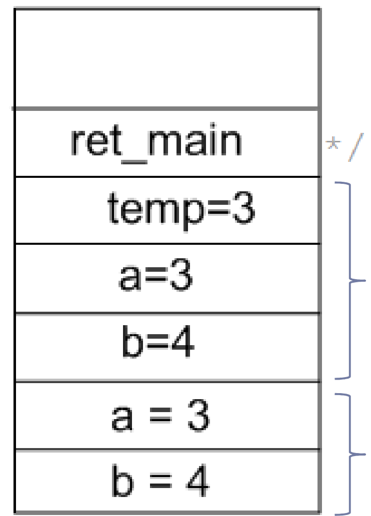
```
#include <iostream>

using namespace std;

void intercambia_1(int , int );


int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_1(a,b);
    cout << "Con intercambia_1 "<<"a = " << a << " b= " << b << endl;

    return 0;
}
/*Función intercambia con paso de parámetros por valor*/
void intercambia_1(int a, int b)
{
    int temp;

    temp=a;
    a=b;
    b= temp;
}
```

Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

```
#include <iostream>

using namespace std;

void intercambia_1(int , int );


int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_1(a,b);
    cout << "Con intercambia_1 "<<"a = " << a << " b= " << b << endl;

    return 0;
}
/*Función intercambia con paso
void intercambia_1(int a, int
{
    int temp;

    temp=a;
    a=b;
    b= temp;
}
```

ret_main

sp



Párametros y variables locales a intercambia_1

Variables locales a main

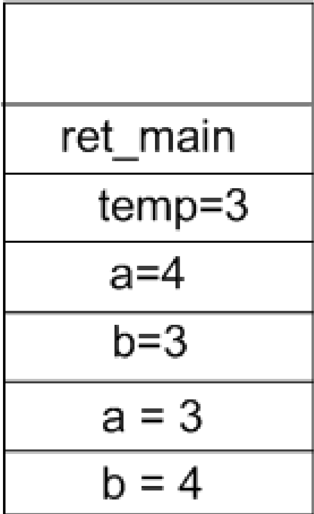
Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

```
#include <iostream>

using namespace std;

void intercambia_1(int , int );


int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_1(a,b);
    cout << "Con intercambia_1 "<<"a = " << a << " b= " << b << endl;

    return 0;
}
/*Función intercambia con paso                       */
void intercambia_1(int a, int
{
    int temp;

    temp=a;
    a=b;
    b= temp;
}
```

El intercambio se ha realizado correctamente a nivel local de la función intercambia_1; pero...

ret_main

sp



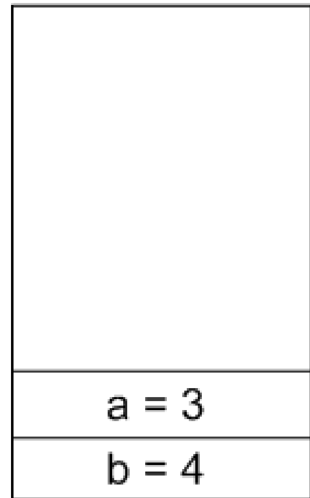
Párametros y variables locales a intercambia_1

Variables locales a main

Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

```
#include <iostream>

using namespace std;

void intercambia_1(int , int );


int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_1(a,b);
    cout << "Con intercambia_1 "<<"a = " << a << " b= " << b << endl;

    return 0;
}
/*Función intercambia con paso                     */
void intercambia_1(int a, int b)
{
    int temp;

    temp=a;
    a=b;
    b= temp;
}
```

...pero cuando la función retorna, esos valores desaparecen y las variables originales de main no han sido modificadas. Sólo se podría devolver un valor mediante return (en el caso de que la función no "devolviera" void)



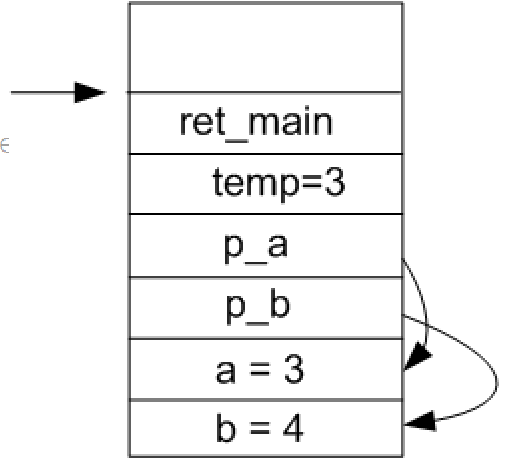
Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

```
#include <iostream>

using namespace std;

void intercambia_punt(int *, int *);

int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_punt(&a,&b);
    cout << "Con intercambia_punt "<<"a = " << a << " b= " << b << endl;

    return 0;
}

/*Función intercambia con paso de parámetros por punte
void intercambia_punt(int *p_a, int *p_b)
{
    int temp;

    temp=*p_a;
    *p_a=*p_b;
    *p_b= temp;
}
```

ret_main

Para resolver este problema en C se utilizan los punteros:

- Se llama a la función con la dirección de las variables (operador &)
- En la funció se accede al valor apuntado por el puntero mediate el operado *



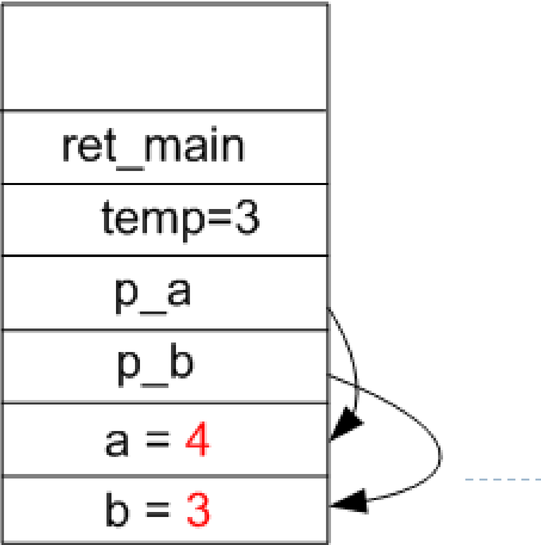
Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

Para resolver este problema en C se utilizan los punteros:

- Se llama a la función con la dirección de las variables (operador &)
- En la funció se accede al valor apuntado por el puntero mediate el operado *

```
#include <iostream>

using namespace std;

void intercambia_punt(int *, int *);

int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_punt(&a,&b);
    cout << "Con intercambia_punt "<<"a = " << a << " b= " << b << endl;

    return 0;
}

/*Función intercambia con paso de parámetros 
void intercambia_punt(int *p_a, int *p_b)
{
    int temp;

    temp=*p_a;
    *p_a=*p_b;
    *p_b= temp;
}
```

ret_main



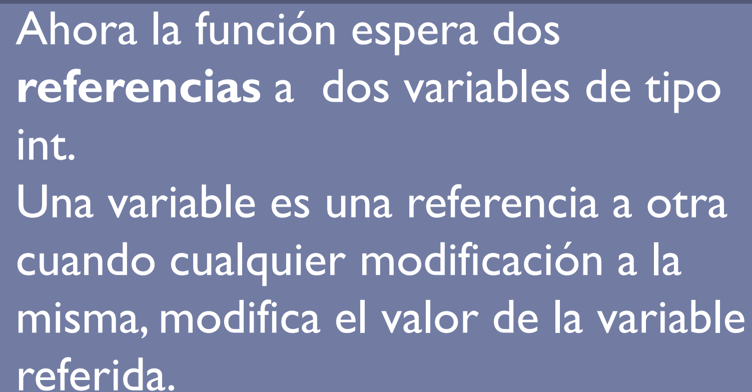
Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

De manera que el **paso por puntero o por dirección** resuelve efectivamente el problema de **devolver más de un valor desde una función**.

El **paso por puntero** ha sido tradicionalmente criticado por la **dificultad que su notación** implica para los programadores.

En C++ se establece el concepto de **paso por referencia** con una notación **más clara** y menos propensa a errores.

```
void intercambia_ref(int &, int &);
```



Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

```
#include <iostream>

using namespace std;

void intercambia_ref(int &, int &);

int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_ref(a,b);
    cout << "Con intercambia_punt "<<"a = " << a << " b= " << b << endl;
    return 0;
}

/*Función intercambia con paso de parámetros por referencia*/
void intercambia_ref(int &r1,int &r2)
{
    int temp;

    temp=r1;
    r1=r2;
    r2= temp;
}
```

Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

```
#include <iostream>

using namespace std;

void intercambia_ref(int &, int &);

int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_ref(a,b);
    cout << "Con intercambia_punt "<<"a = " << a << " b= " << b << endl;
    return 0;
}

/*Función intercambia con paso de parámetros por referencia*/
void intercambia_ref(int &r1,int &r2)
{
    int temp;

    temp=r1;
    r1=r2;
    r2= temp;
}
```

Se pasa directamente el valor de las variables que van a recibir el resultado de la función intercambia_ref

Se trabaja directamente con los parametros formales (que en este caso contendrá las refencias a las variables **a** y **b** de main). La solución es mucho más clara.

Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

```
#include <iostream>

using namespace std;

void intercambia_ref(int &, int &);

int main()
{
    int a,b;
    cout << "Diga los valores a intercambiar" << endl;
    cin >> a >> b;

    intercambia_ref(a,b);
    cout << "Con intercambia_punt "<<"a = " << a << " b= " << b << endl;
    return 0;
}

/*Función intercambia con paso de parámetros por referencia*/
void intercambia_ref(int &a,int &b)
{
    int temp;

    temp=a;
    a=b;
    b= temp;
}
```

Se pasa directamente el valor de las variables que van a recibir el resultado de la función intercambia_ref

Se trabaja directamente con las referencias

Paso de parámetros a las funciones. Paso por referencia (C++) vs. paso por puntero (C)

```
#include <iostream>

using namespace std;

struct estructura_muy_grande
{
    int arr[1000];
};

void rellena(estructura_muy_grande &est)
{
    for (int i=0; i < 1000;i++)
        est.arr[i]=i;
}
void muestra(const estructura_muy_grande &est)
{
    for (int i=0; i < 1000;i++)
        cout << est.arr[i] << " ";
    cout << endl;
}
int main()
{
    estructura_muy_grande  m;

    rellena(m);

    muestra(m);
}
```

A veces se pasa parámetro por referencia por eficiencia aunque no se pretenda modificar en la función. En ese caso es buena práctica definir parámetro como constante.

Aquí la referencia es imprescindible porque se modifica el dato

En este caso se pasa por referencia por eficiencia porque la función no se modifica. Observe el uso de **const** para asegurar que este sea el caso.

Paso de parámetros a las funciones. Paso por movimiento (C++ v11)

El paso por referencia no funciona con objetos temporales (rvalues)

```
#include <iostream>

using namespace std;


void muestra(int& est)
{

        cout << est << endl;

}
int main()
{

    muestra(2+5);
}
```

Error!

```
#include <iostream>

using namespace std;


void muestra(int&& est)
{

        cout << est << endl;

}
int main()
{

    muestra(2+5);
}
```

&& se puede pasar objetos temporales de forma muy eficiente, sin hacer una copia como en el caso del paso por valor

Funciones. Sobrecarga de funciones

En C dos *funciones diferentes no pueden tener el mismo nombre*.

En C++ se pueden tener tantas funciones diferentes con igual nombre como se quiera, siempre que difieran en, al menos, el tipo de un parámetro de la lista de parámetros formales.

En este caso se dice que la función en cuestión está **sobrecargada** (*overloaded*).

El compilador decide que función utilizar en base a la lista de parámetro reales.

```
void intercambia(int *, int *);
void intercambia(int &, int &);
```

Ej: Utilice ambos prototipos anteriores en el mismo programa.

Diseñe otros ejemplo de sobrecarga de funciones.

Funciones. Parámetros por defecto.

El C++ brinda la posibilidad de especificar en la definición de una función el valor que tomarán por defecto ciertos parámetros si la función es llamada sin sumnistrar valores para los mismos.

```
#include <iostream>

using namespace std;

void porDefectoPrueba(int a, int b=1, int c=2);

int main()
{
    porDefectoPrueba(3,2);
    porDefectoPrueba(5);
    return 0;
}

void porDefectoPrueba(int a, int b, int c)
{
    cout << "a= " << a << " b: " << b << " c: " << c << endl;
}
```

Funciones. Parámetros por defecto.

```
#include <iostream>

using namespace std;

void porDefectoPrueba(int a, int b=1, int c=2);

int main()
{
    porDefectoPrueba(3,2);
    porDefectoPrueba(5);
    return 0;
}

void porDefectoPrueba(int a, int b, int c)
{
    cout << "a= " << a << " b: " << b << " c: " << c << endl;
}
```

En la declaración se establecen los valores por defecto que tendrán los parámetros. Los parámetros por defecto deben ocupar las posiciones más a la derecha.

Al llamar a la función, si se omiten los parámetros, estos tendrá el valor declarado por defecto. Si se omite un parámetro, todos los de su derecha se tiene que omitir también: porDefecto(3,,5) Error!!!

Informática Industrial

Práctica 3. Desarrollo de clase simple. Strings. Streams.
Manejo de errores estructurados.

Del C al C++

- Elementos de la librería estándar C++
  - Cadenas de caracteres (clase ***string***)
  - Ficheros
  - Manejo de excepciones (errores)

Del C al C++

Las **clases** en C++ constituyen una forma de **crear nuevos tipos** que estén en pie de igualdad con los tipos propios del lenguaje (**int**, **char**, **float** ...).

Resultan una extension del concepto de estructura en C (**struct**).

Pero además de contener campos que contienen datos, poseen funciones miembro, llamadas **métodos**.

Del C al C++

Estructuras en C:

```
struct vectorPlano
{
    double x;
    double y;
};

int main()
{
    struct vectorPlano v1= {1,1}, v2={2,2},suma;
    double tam;

    tam=sqrt(v1.x*v1.x + v2.y*v2.y);
    suma.x=v1.x+v2.x;
    suma.y= v1.y+v2.y;
}
```

Del C al C++

Estructuras en C:

```
struct vectorPlano
{
    double x;
    double y;
};

int main()
{
    struct vectorPlano v1= {1,1}, v2={2,2},suma;
    double tam;

    tam=sqrt(v1.x*v1.x + v2.y*v2.y);
    suma.x=v1.x+v2.x;
    suma.y= v1.y+v2.y;
}
```

Definición del nuevo tipo de dato. Contiene dos campos de tipo double: datos.

Definimos tres variables de ese nuevo tipo de datos. En C++ podríamos no haber puesto la palabra **struct** en la definición de las variables.

Se calcula la longitud de un vector (accediendo a sus miembros) y otro vector que sea la suma de los dos anteriores. No hay una relación directa entre el tipo de dato y las funciones o expresiones que actúan sobre ese tipo de dato.

Del C al C++

```
#include <iostream>
#include <math.h>
using namespace std;

class VectorPlano
{
    public:
        double  x;
        double  y;
        VectorPlano()
        {
            x=0; y=0;
        }
        VectorPlano(double xi, double yi)
        {
            x=xi;y=yi;
        }
        double norma()
        {
             return sqrt(x*x+y*y);
        }
        friend VectorPlano operator+(const VectorPlano &a, const VectorPlano &b)
        {
            return VectorPlano(a.x+b.x,a.y+b.y);
        }
};
```

Del C al C++ Clases en C++:

```
#include <iostream>
#include <math.h>
using namespace std;

class VectorPlano
{
    public:
        double  x;
        double  y;
        VectorPlano()
        {
            x=0;  y=0;
        }
        VectorPlano(double xi, double yi)
        {
            x=xi;y=yi;
        }
        double norma()
        {
             return sqrt(x*x+y*y);
        }
        friend VectorPlano operator+(const VectorPlano &a, const VectorPlano &b)
        {
            return VectorPlano(a.x+b.x,a.y+b.y);
        }
};
```

Los detalles serán explicadas en una práctica posterior. Por lo pronto notad que existen miembros de tipo dato, como en las estructuras de C y métodos.

Los métodos (sobrecargados) con el mismo nombre que la clase, son los constructores, llamados a la hora de la creación de los objetos de esa clase.

Del C al C++

Clases en C++:

```
#include <iostream>
#include <math.h>
using namespace std;

class VectorPlano
{
    public:
        double  x;
        double  y;
        VectorPlano()
        {
            x=0; y=0;
        }
        VectorPlano(double xi, double yi)
        {
            x=xi;y=yi;
        }
        double norma()
        {
             return sqrt(x*x+y*y);
        }
        friend VectorPlano operator+(const VectorPlano &a, const VectorPlano &b)
        {
            return VectorPlano(a.x+b.x,a.y+b.y);
        }
};
```

Sabemos como opera la + en tipos básicos (int, float). En el caso de que tenga sentido, este comportamiento puede se asumido en las clases definidas por el programador, **sobrecargando** el operador concreto que se va a aplicar (+)

Del C al C++

Clases en C++:

Se llama al constructor correspondiente. Qué valor inicial tendrán los campos x e y de c.

```
int main()
{
    VectorPlano a(1,1),b(2,2),c;

    c=a+b;
    cout << "Norma a: " << a.norma() << endl;
    cout << "x de c: " << c.x << " y de c:" << c.y << endl
    return 0;
}
```

El operador suma ha sido definido para estas clases. Suma los elementos correspondientes

Se accede al método. Note paréntesis (es una función).

Se accede a miembros tipo de datos.

Del C al C++

Clases en C++:

Se llama al constructor correspondiente. Qué valor inicial tendrán los campos x e y de c.

```
int main()
{
    VectorPlano a(1,1),b(2,2),c;

    c=a+b;
    cout << "Norma a: " << a.norma() << endl;
    cout << "x de c: " << c.x << " y de c:" << c.y << endl
    return 0;
}
```

El operador suma ha sido definido para estas clases. Suma los elementos correspondientes

Se accede al método. Note paréntesis (es una función).

Se accede a miembros tipo de datos.

Del C al C++

```
class VectorPlano
{

        double  m_x;
        double  m_y;
    public:
        VectorPlano()
        {
            m_x=0; m_y=0;
        }
        VectorPlano(double xi, double yi)
        {
            m_x=xi;m_y=yi;
        }

        double GetX() {return m_x;}

        double GetY() {return m_y;}

        double norma()
        {
             return sqrt(m_x*m_x+m_y*m_y);
        }
        friend VectorPlano operator+(const VectorPlano &a, const VectorPlano &b)
        {
            return VectorPlano(a.m_x+b.m_x,a.m_y+b.m_y);
        }
         friend ostream&  operator<<(ostream &out, VectorPlano &v)
        {
            out << "(" << v.m_x <<"," << v.m_y <<")";
            return out;
        }
};
```

```
int main()
{
    VectorPlano a(1,1),b(2,4),c;

    c=a+b;
    cout << "Norma a: " << a.norma() << endl;
    cout << "x de c: " << c.GetX() << " y de c:" << c.GetY() << endl;
    cout << b << endl;
    return 0;
}
```

Sobrecarga del operador << para utilizar la salida estándar

Del C al C++. Clase string.

- Primero veremos como usar las clases que vienen de forma **estándar** con el sistema para después considerar en más detalle como crear nuestras clases propias.
- El C++ provee un nuevo tipo de datos (implementado como una clase) para trabajar con las **cadenas de caracteres**.
- El uso de este nuevo tipo de datos (**string**) supone una serie de ventajas con respecto al *array* de *chars* utilizado en C. Especialmente el uso de **string** permite que no tengamos que *preocuparnos* excesivamente del tamaño de la cadena.

Del C al C++. Clase string.

- Ej: Con cadenas de caracteres de C:

```
#include <iostream>
#include <string.h>

using namespace std;

int main()
{
    char prefijo[6]="pre", palabra[30]="universitario";
    char compuesto[40];

    strcpy(compuesto,prefijo);
    strcat(compuesto,palabra);

    cout << "La palabra compuesta es " << compuesto << "\n";
}
```

Hay que incluir el viejo fichero de cabecera de C que define **strcpy** y **strcat** entre otras. Observe que se pone extensión .h

Tenemos que reservar espacio explícitamente para cada cadena.

Utilizar las funciones de la biblioteca estándar que esperan un array de char terminado en '\0'

Del C al C++. Clase string.

- Ej: Con clase string de C++:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string prefijo="pre", palabra="universitario";
   string compuesto;

   compuesto = prefijo + palabra;

   cout << "La palabra compuesta es " << compuesto << "\n";
}
```

Incluimos la cabecera de C++. (sin extensión) Las definiciones de **string** se encuentran en el namespace std

Definimos objetos string y les asignamos valores iniciales. No hace falta especificar tamaño

Se utilizan los operadores = y + para las operaciones de copia concatenación de cadenas. Se dice que dichos operadores están **sobrecargados**. No hay que preocuparse del tamaño de la cadena compuesta: los objetos string crecen de tamaño para acomodar la cadena resultante.

Del C al C++. Clase string.

- Los objetos de la clase string poseen una serie de **métodos** útiles para trabajar con cadenas de caracteres:

```
#include <iostream>

using namespace std;

int main()
{

    string g("Cadena inicial");

    /*Comenzando en el caracter 6 de la cadena original,
    reemplaza 1 caracter con los 15 caracteres de la cadena que se da*/
    cout << g.replace(6,1," de caracteres ",15) << endl;

    /*Comenzando en el caracter 7 de la cadena original,
    reemplaza los 7 caracteres sigguientes con los 5 caracteres de la cadena que se da*/
    string h("Cadena inicial");

    cout << h.replace(7,7,"final",5) << endl;


    return 0;
}
```

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string msg;

    msg= "Se asigna una cadena utilizando signo = (sobrecargado)";

    cout << msg << "longitud de cadena " << msg.size() << endl;



    int pos=msg.find("asigna");

    cout << "La primera ocurrencia de asigna en la cadena es en pos "<< pos << endl;

    string sub;

    sub = msg.substr(pos,msg.size());

    cout << "La subcadena que comienza en "<< pos << " es: " << sub << endl;

    cout << "Caracteres individuales se acceden con sitanxis de array. El primero es:  "<< msg[0] << endl;
    return 0;
}
```

Del C al C++. Clase string.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string palabra ("fin");
    string adivina;

    do
    {
        cout << "Adivina palabra de " << palabra.size() << " letras." << endl;
        cin >> adivina;
        if (adivina < palabra)
             cout << "La palabra que has dado es menor" << endl;
        if (adivina > palabra)
             cout << "La palabra que has dado es mayor" << endl;
    } while (adivina != palabra);


}
```

Los operadores de comparación han sido *sobrecargados* por la clase **string** de manera que se pueden utilizar. ¿Recuerdas como se hacía en C?

Del C al C++. Clase string.

```
#include <iostream>
#include <string>
#include <string.h>

using namespace std;

int main()
{
    string msg;
    char msg_old[130];

    msg="Esta es una cadena de caracteres\n";

    strcpy(msg_old,msg.c_str());

    cout <<  msg_old << endl;

    return 0;
}
```

Método c_str devuelve un puntero a una cadena de estilo C. Util para utilizar funciones que esperan ese tipo de cadenas.

Del C al C++. Streams. E/S.

- Un *stream* o flujo de datos es una **serie de caracteres** hacia o desde un medio de almacenamiento (ej: fichero).
- Los ficheros constituyen una forma de hacer que los datos sean persistentes: se graben ("*sobrevivan*") de una ejecución a otra del mismo programa o sirvan para intercambiar información entre varios programas que conozcan la estructura del fichero.
- Ya hemos trabajado con ***streams*** al hacer la e/s desde/ hacia la consola (***cin***, ***cout***). Se trata de generalizar estos conceptos para ficheros de texto.

Del C al C++. Streams. E/S.

- ¿Cómo se realiza la e/s desde/hacia un fichero? En realidad se maneja como un serie de caracteres que se envía o reciben uno, detrás del otro?

'T'	'e'	'm'	'p'		'2'	'9'	'.'	'5'	

- Pero ¿cómo se sabe si corresponden a una frase o a un número? Y si es un número ¿de que tipo: real o entero?
- Es el programador el que decide esto, al imponer o conocer la estructura del fichero.
- La naturaleza serie del stream es importante: no puedes hacer un acceso al 7º elemento del ejemplo anterior y concluir que se trata de un 9. Para poder extraer el valor correcto (real) de 29.5 hay que leer desde el principio.

Del C al C++. Streams. E/S.

- Ejemplo de fichero de salida:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
        ofstream salida("MiFichero.txt",ios::app);
        if(salida.good())
        {
                salida <<"Este texto se añade al final." << endl;
                salida.close();
        }
        else
            cerr << "Error en la apertura del fichero" << endl;
        return 0;
}
```

Del C al C++. Streams. E/S.

- Ejemplo de fichero de salida:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
        ofstream salida("MiFichero.txt",ios::app);
        if(salida.good())
        {
                salida <<"Este texto se añade al final." << endl;
                salida.close();
        }
        else
            cerr << "Error en la apertura del fichero" << endl;
}
```

Cabecera para los streams a ficheros

Declaro un objeto (llamado *salida*) que pertenece a la clase **ofstream** indicando que se trata un fichero de salida. Durante la inicialización paso nombre de fichero y el modo de apertura

Los modos de apertura vienen definidos en el namespace ios.

Método good() devuelve TRUE si se ha abierto de manera correcta.

Del C al C++. Streams. E/S.

- Ejemplo de fichero de salida:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
        ofstream salida("MiFichero.txt",ios::app);
        if(salida.good())
        {
                salida <<"Este texto se añade al final." << endl;
                salida.close();
        }
        else
            cerr << "Error en la apertura del fichero" << endl;
        return 0;
}
```

La salida se realiza a partir de la sobrecarga del operador << igual que en ***cout.***

Reporte de error deben realizarse al dispostivo **cerr** que normalmante se asocia a la pantall.

Método **close()** para cerrar apropiadamente el ***stream***.

Del C al C++. Streams. E/S.

- Algunas clases relacionadas con I/O ficheros:

Clase	Propósito
fstream	Tanto entrada como salida
ofstream	Stream de salida (escritura)
ifstream	Stream de entrada (entrada)

- Modos de apertura más importantes:

Modo	Propósito
ios::app	Append: operación de salida al final del fichero existente.
ios::in	Permite operaciones de entrada
ios::out	Permite operaciones de salida

Del C al C++. Streams. E/S.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
   ifstream entrada;
   float temp;

   entrada.open("Temperaturas.txt", ios::in);
   if (entrada.good())
   {
       while (!entrada.eof())
       {
           entrada >> temp;
           if (!entrada.fail())
               cout << "Temp: " << temp << endl;
       }
    entrada.close();

   }
   else
     cerr << "Error de apertura de fichero" << endl;
}
```

Del C al C++. Streams. E/S.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
   ifstream entrada;
   float temp;

   entrada.open("Temperaturas.txt", ios::in);
   if (entrada.good())
   {
       while (!entrada.eof())
       {
           entrada >> temp;
           if (!entrada.fail())
                cout << "Temp: " << temp << endl;
       }
    entrada.close();

   }
   else
     cerr << "Error de apertura de fichero" << endl;
}
```

Se define el fichero de entrada. Esta vez de inicio no se ha asociado a ningún fichero.

Se abre el fichero de entrada. mediante método **open()** y se comprueba si se ha tenido éxito

Del C al C++. Streams. E/S.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
   ifstream entrada;
   float temp;

   entrada.open("Temperaturas.txt", ios::in);
   if (entrada.good())
   {
       while (!entrada.eof())
       {
           entrada >> temp;
           if (!entrada.fail())
               cout << "Temp: " << temp << endl;
       }
    entrada.close();

   }
   else
     cerr << "Error de apertura de fichero" << endl;
}
```

Mientras no se llegue al final del fichero (método eof())

Se extrae el siguiente elemento del stream que sea compatible con un ***float*** (la estructura del fichero la decide el programador)

Si la sucesión serie de caracteres se ha podido interpretar como float (vea uso de método fail()), se saca valor por pantalla.

Del C al C++. Manejo de errores.

- Durante la ejecución de un programa pueden ocurrir errores o circunstancias excepcionales. Estas situaciones deben ser previstas y tratadas por el programador.
- En C, la forma habitual de hacer lo anterior, es mediante el uso de condicionales que hay que desplegar a lo largo de todo el código y que mezcla la detección con el tratamiento del error, dificultando la comprensión del programa.

```
//Realiza alguna función no especificada y
//devuelve código de error en los casos apropiados
//0 - no error
//no cero - número del error
int funcion();

int main()
{
    int err;

    err = funcion();
    if (err!=0)
    {
        printf("Error tipo %d\n", err);
        //Otros tratamientos del error
    } else {
        printf("No error\n");
    }


    return 0;
}
```

Se prevee la posibilidad de que, como consecuencia del funcionamiento de la función, esta devuelva un código de error cuyo tratamiento debe ser contemplado en el programa por la función que la utiliza.

Del C al C++. Manejo de errores.

- El C++ incorpora un novedoso mecanismo para tratar estos errores (**excepciones**) que puedan surgir durante la **ejecución** del programa (***exception handling***).
- Ofrece un mecanismo mediante el cual, si ocurre un error, se tranfiere el control a un manipulador del error (*handler*) previamente definido.
- Evita la necesidad de devolver código de error desde las funciones porque el nuevo mecanismo **propaga los errores o excepciones a través de varias llamadas a funciones** haciendo innecesaria la existencia de código específico para tratar el error es las llamadas intermedias.

```
try
{
    //código suceptible
    //de generar errores
    throw  (type) param_real
}
catch (type param_formal)
{
    //Tratamiento de los errores de tipo type
}
catch (...)
{
    //Captura errores de cualquier tipo
}
```

Del C al C++. Manejo de errores.

```
try
{
    //código suceptible
    //de generar errores
    throw  (type) param_real
}
catch (type param_formal)
{
    //Tratamiento de los errores de tipo type
}
catch (...)
{
    //Captura errores de cualquier tipo
}
```

En la evaluación del bloque **try**, en caso de ocurrir un error de ejecución, se lanzará una excepción (**throw**).
Esta excepción se lanzará directamente dentro del bloque **try** o por alguna función llamada dentro de ese bloque.

El bloque **catch** da tramiento a los errores de tipo *type* que se hayan producido dentro del bloque **try**. Puede haber tantos bloque **catch** como se quiera, siempre que los tipos de los errores difieran.

Del C al C++. Manejo de errores.

```
#include <iostream>

using namespace std;

const int DivideCero=1;
float divide(float dividendo, float divisor);

int main()
{
    float dividendo,divisor;

    cout << "Dividendo y divisor" << endl;
    cin >> dividendo >> divisor;
    try
    {
        cout << "Resultado: " << divide(dividendo,divisor) << endl;
    }
    catch (int ex)
    {
        if (ex == DivideCero)
            cerr << "Error de división por cero" << endl;
    }
    cout << "En ambos casos, se llega aqui" << endl;
}
float divide(float dividendo, float divisor)
{
    if (divisor == 0)
    {
        throw DivideCero;
    }
    return dividendo/divisor;
}
```

Esta función que se llama dentro del **try** puede generar un error.

Bloque **catch** se activa cuando se genera error de tipo entero.

Se genera un error de tipo **int**.

Del C al C++. Manejo de errores.

```
#include <iostream>

using namespace std;

const int DivideCero=1;
float divide(float dividendo, float divisor);
float Otra(float,float);
int main()
{
    float dividendo,divisor;

    cout << "Dividendo y divisor" << endl;
    cin >> dividendo >> divisor;
    try
    {
        //cout << "Resultado: " << divide(dividendo,divisor) << endl;
        cout << "Resultado: " << Otra(dividendo,divisor) << endl;
    }
    catch (int ex)
    {
        if (ex == DivideCero)
            cerr << "Error de división por cero" << endl;
    }
    cout << "En ambos casos, se llega aqui" << endl;
}
float Otra(float dividendo,float divisor)
{
    return divide(dividendo,divisor);
}
```

Ahora el error se sigue generando como antes pero hay **una función intermedia**. El mecanismo de manejo de errores sigue funcionado sin cambios...

Informática Industrial

Práctica 4. Arrays estáticos y dinámicos. Indrotucción a clases y algoritmos STL.

Del C al C++

- Reserva dinámica de memoria en C++.
- Trabajo con *arrays*.
- Algoritmos de búsqueda y ordenación.
- Algunos conceptos de eficiencia computacional.
- Elementos de programación genérica.

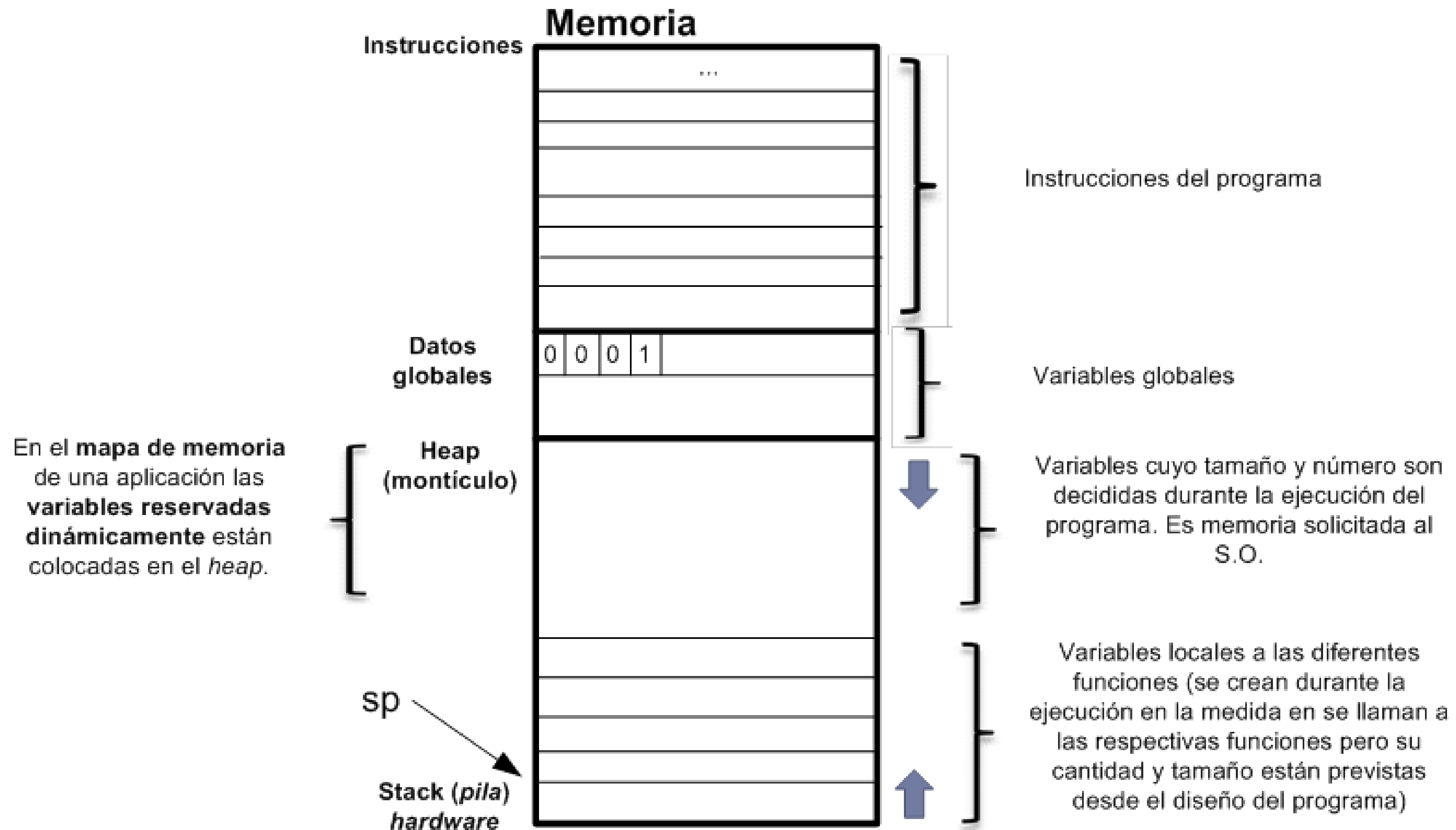
Del C al C++

- Reserva dinámica de memoria en C++.

Son variables cuya existencia se decide "dinámicamente" durante la ejecución del programa.

Esto a diferencia de las otras variables vistas hasta ahora (globales o locales) que se deciden desde el diseño y compilación del programa.

Del C al C++. Reserva dinámica de memoria en C++.



Del C al C++

- Reserva dinámica de memoria en C++.

En C el manejo dinámico de memoria se realizaba mediante a llamadas a funciones (***malloc***(), ***free*** ()) que a su vez hacían las llamadas correspondientes a los servicios del Sistema Operativo.

En C++ el manejo dinámico de memoria está integrado directamente en el lenguaje.

Se utilizan las palabras reservadas:

new: para reservar memoria

delete: para liberarla

Del C al C++

- Reserva dinámica de memoria en C++.

```
//Ejemplo para una sola variable

int *p;

p = new int;
//p es un puntero a una variable reservada dinámicamente de tipo int
*p=10;

...
delete p;
```

Pero la reserva dinámica es útil sobre todo para reservar memoria para un *array*. Para ello se utiliza: new *type* [tam] y delete [ ]

Del C al C++

- Reserva estática vs. reserva dinámica

```
#include <iostream>

using namespace std;

int main()
{
    int vector[100];
    int i;

    for (i=0;i < 100 ; i++)
    {
        cout << "Diga valor de elemento " << i << endl;
        cin >> vector[i];
    }

    return 0;
}
```

Se define el vector y se decide en tiempo de compilación el tamaño del vector

Del C al C++

- Reserva estática vs. reserva dinámica

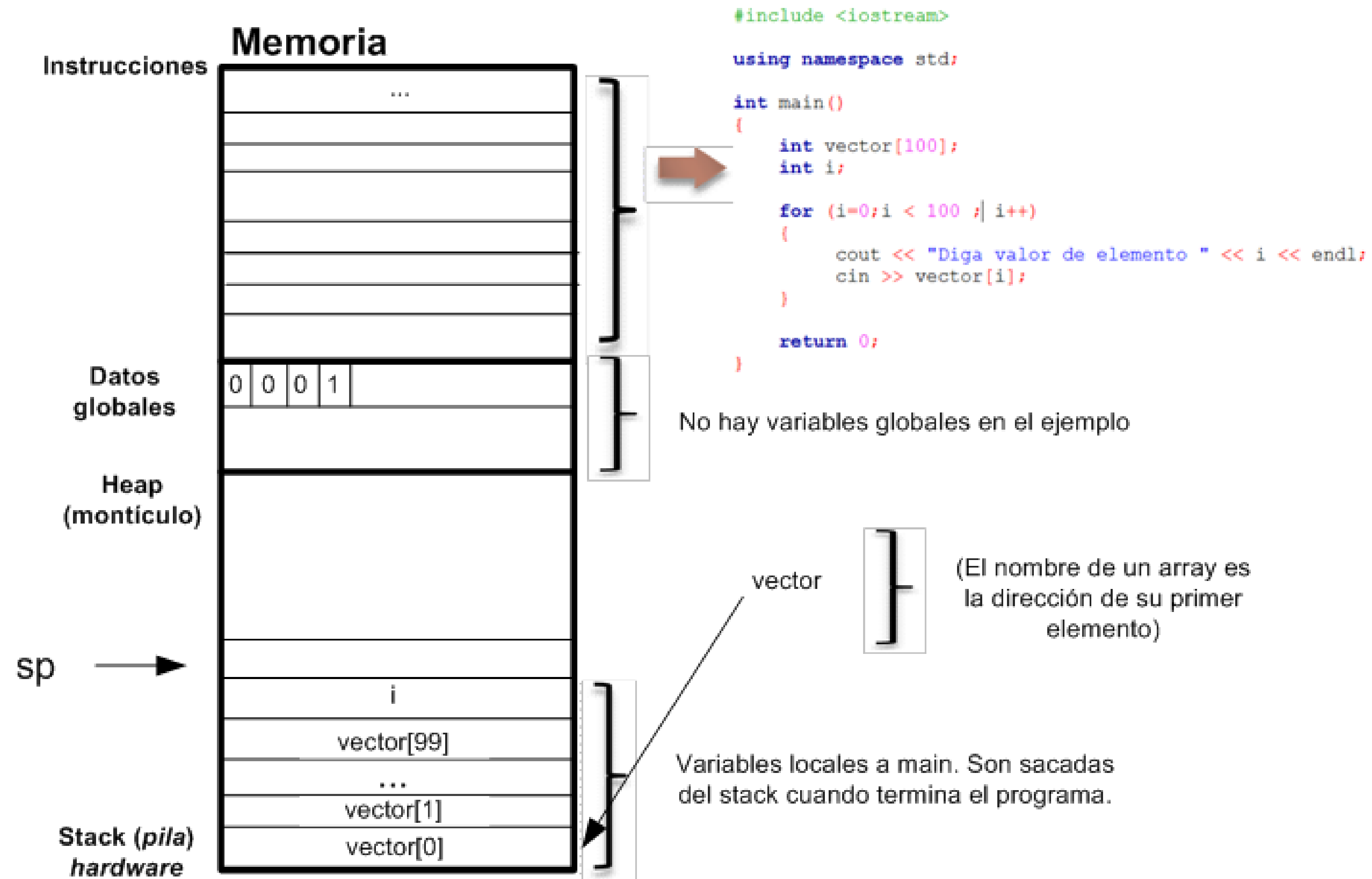
```
#include <iostream>

using namespace std;

int main()
{
    int *vector;
    int tam,i;

    cout << "Diga el tamaño del vector" << endl;
    cin >> tam;

    vector= new int[tam];

    for (i=0;i < tam ; i++)
    {
        cout << "Diga valor de elemento " << i << endl;
        cin >> vector[i];
    }
    delete [] vector;
    return 0;
}
```

Se define un puntero al tipo de datos que sea (int). El puntero contendrá la dirección de la memoria asignada dinámicamente.

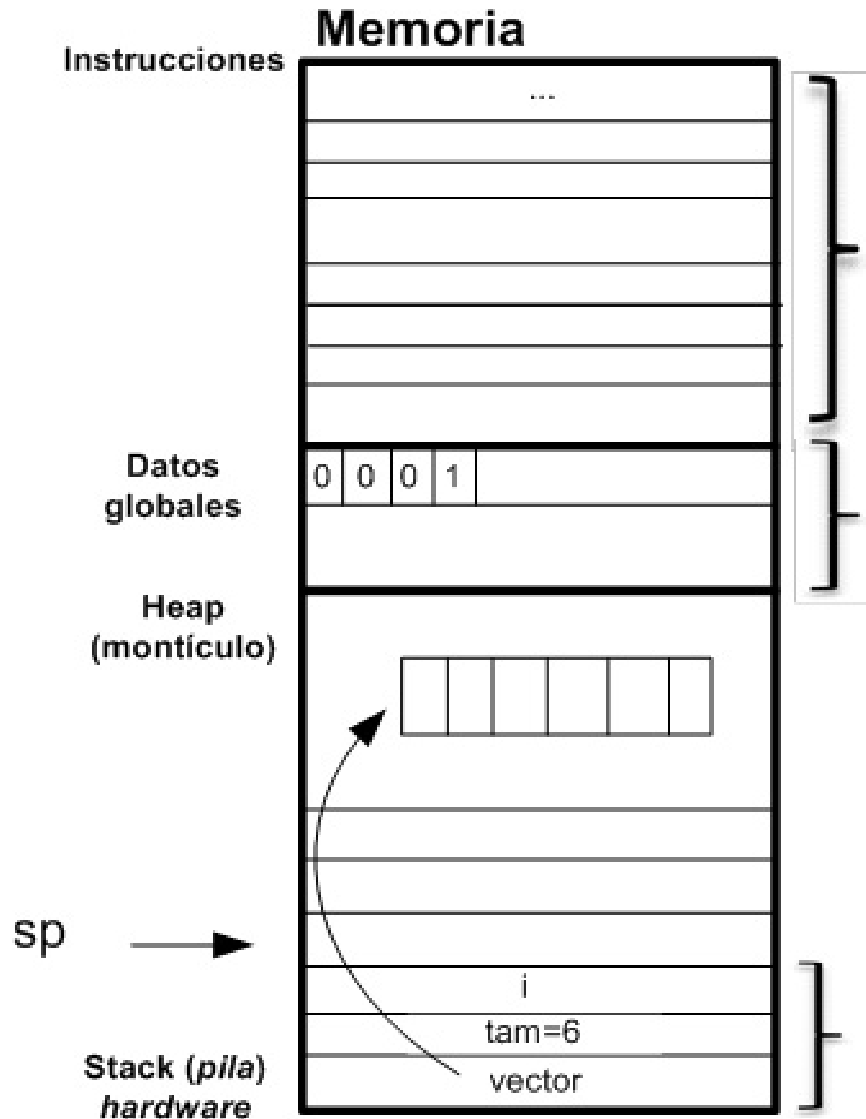
Se obtiene el tamaño del vector y se solicita la cantidad de memoria requerida mediante **new.**

Es buena práctica liberar (**delete o delete [ ] para vectores**) la memoria asignada dinámicamente.

Del C al C++



Del C al C++




```
#include <iostream>

using namespace std;

int main()
{
    int *vector;
    int tam, i;

    cout << "Diga el tamaño del vector" << endl;
    cin >> tam;

    vector= new int[tam];

    for (i=0;i < tam ; i++)
    {
        cout << "Diga valor de elemento " << i << endl;
        cin >> vector[i];
    }

    return 0;
}
```

No hay variables globales en el ejemplo

Variables locales a main. Son sacadas del stack cuando termina el programa.

Reserva dinámica. Tratamiento de error.

Pero ¿qué pasa si el S.O operativo no tiene suficiente memoria para servir la petición que se le hace durante la ejecución de programa?

Ocurriría un error de tiempo de ejecución que debe ser tratado.

Para ello existen dos posibilidades.

Del C al C++

- Reserva dinámica. Tratamiento de error. Variante 1

```
#include <iostream>

using namespace std;

int main()
{
    int *vector;
    int tam;


    vector= new (std::nothrow) int[(int) 1e10];
    if (vector == nullptr)
    {
        cerr << "Error de reserva dinámica" << endl;
    }
    else
    {
        for (int i=0; i < tam ; i++)
        {
            cout << "Diga valor de elemento " << i << endl;
            cin >> vector[i];
        }
        delete [] vector;
    }

    return 0;
}
```

Uso de *nothrow* para que no se lance una excepción.

Trabajamos con el vector sólo si el valor devuelto es diferente de **NULL**.

Del C al C++

- Reserva dinámica. Tratamiento de error. Variante 2

```
#include <iostream>
#include <exception>

using namespace std;

int main()
{
    int *vector;
    int tam;

    try
    {
        vector= new int[(int) 1e10];
    }

    catch (exception &e)
    {
        cerr << e.what() << endl;
        return -1;
    }

    for (int i=0; i < tam ; i++)
    {
        cout << "Diga valor de elemento " << i << endl;
        cin >> vector[i];
    }
    delete [] vector;


    return 0;
}
```

Uso del sistema de manejo de errores. Con error predefinido. Objeto de tipo exception definido en <exception>

Se pasa la excepción por referencia. Contiene el método what() que explica el error.

Arrays.

Los arrays son estructuras muy utilizadas. Las características que los definen son:

- Conjunto de datos del **mismo tipo**.
- Colocados **contiguamente en memoria**.
- El **nombre** del array es un **puntero** al primer elemento del mismo.

A partir de esta definición el proceso de acceso a un elemento determinado, dado por su indice, es una operación muy rápida.

Ej: int a[10];

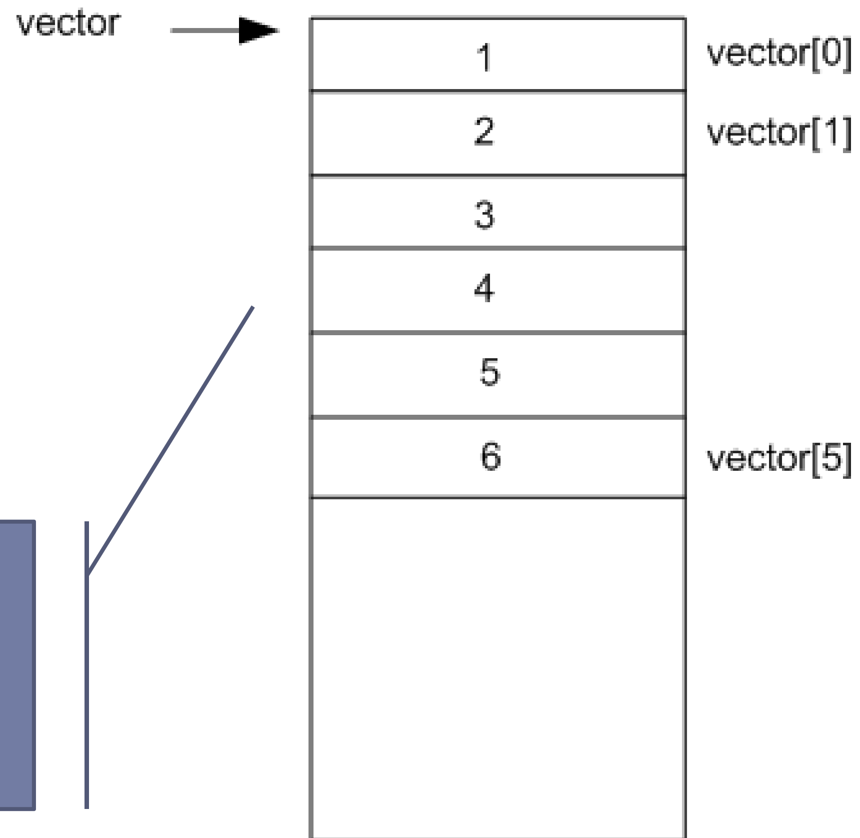
```
a[5]=10;
```

Las operaciones de inserción y borrado de un elemento son operaciones más complejas. ¿Por qué?

Del C al C++

- *Arrays.*

```
#include <iostream>

using namespace std;

int main()
{
    int vector[6]={1,2,3,4,5,6};
    int tam=6;


    return 0;
}
```

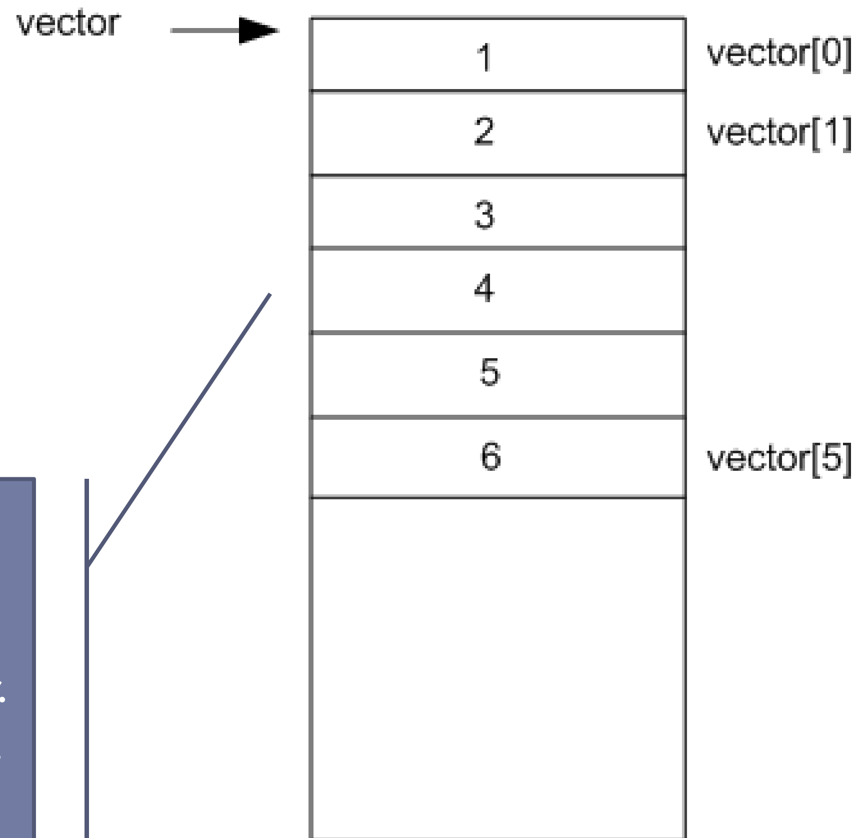


¿Qué hacer si queremos borrar el cuarto elemento?

Del C al C++

- *Arrays.*

```
#include <iostream>

using namespace std;

int main()
{
    int vector[6]={1,2,3,4,5,6};
    int tam=6;


    return 0;
}
```



¿Qué hacer si queremos borrar el cuarto elemento?

- Mover hacia arriba a lo elementos de índice mayor.
- Decrementar el valor de la variable que recoje el tamaño del vector

Del C al C++

- *Arrays*. Borrando un elemento.

```
#include <iostream>

using namespace std;

int main()
{
    int vector[6]={1,2,3,4,5,6};
    int tam=6, elem;

    cout << "Diga elemento que quiere borrar" << endl;
    cin >> elem;

    if (elem < tam)
    {
        tam--;
        for (int i=elem; i <= tam; i++)
            vector[i]=vector[i+1];

    }

    return 0;
}
```

Arrays. Insertando un elemento en array dinámico.

De forma similar, la inserción de un elemento en un array implica el movimiento en memoria de una cantidad de datos, que para *arrays* grandes, puede ser considerable.

La inserción además tiene la complejidad de que hay que determinar si se tiene suficiente espacio reservado. En los arrays estáticos no se puede insertar si no hay espacio suficiente.

Los arrays dinámicos resuelven este problema. ¿Cómo se realizaría la inserción de un elemento en un array dinámico?

Del C al C++

- *Arrays*. Insertando un elemento en array dinámico.

```
#include <iostream>
using namespace std;

int main()
{

   int tam=10;  //Tamanno inicial a reservar dinámicamen
   int *vector;
   int pos;

   vector=new  int[tam];
   //Inicializamos con valores arbitrarios
   for (int i=0; i < tam; i++)
        vector[i]=i;

   cout << "Posicion en la que desea insertar" << endl;
   cin >> pos;
```

Arrays. Insertando un elemento en array dinámico. (cont)

```
    if (pos <= tam)
    {
        int *aux;

        tam++;
        aux= new int [tam];
        for (int i=0; i < pos; i++)
             aux[i]=vector[i];
        cout << "Diga elemento a insertar " << endl;
        cin >> aux[pos];
        for (int i=pos+1; i < tam; i++)
             aux[i]=vector[i-1];

        delete [] vector;
        vector=aux;
    }
    else{
         cout << "El vector tiene solo " << tam << " elementos." << endl;
    }
     for (int i=0; i < tam; i++)
         cout << vector[i] << " ";
return 0;
}
```

Arrays. Resumen

Los *arrays* dinámicos permiten definir el tamaño durante la ejecución del programa.

En cualquier caso, **el acceso** a un elemento del array dado su índice es una operación cuya **velocidad es constante**: no depende del tamaño del array.

La **inserción** y el **borrado** de un elemento del array (debido a su definición como elementos contiguos en memoria) implica un tiempo de ejecución que, en promedio, depende ***linealmente*** del tamaño del array.

¿Qué otras operaciones ud. conoce se pueden hacer con un array?

Arrays. Resumen

Las **ordenación** y **búsqueda** de un dato en un conjunto dado son operaciones básicas en programación?

Escriba una función que reciba un *array*, de enteros, su tamaño, y el valor de un entero y que devuelva la posición en que aparece ese entero en el array o -1 si no aparece.

Si el array estuviese ordenado, de que otra forma se puede programar para que la búsqueda se más eficiente.

```
int busca(int [], int, int);
```

Del C al C++

- *Arrays*. Búsqueda en array desordenado.

```
#include <iostream>

using namespace std;

int busca(int [], int, int);

int main()
{
    int vector[]={1,4,10,3,15,22,18},pos;

    pos=busca(vector,7,15);
    if (pos >= 0)
        cout << "El elemento aparece en la posición " << pos << endl;
    else
        cout << "El elemento no se encuentra en el vector" << endl;
}

int busca(int v[], int tam, int elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;

}
```

Del C al C++

- *Arrays*. Búsqueda en array desordenado.

```
#include <iostream>

using namespace std;

int busca(int [], int, int);

int main()
{
    int vector[]={1,4,10,3,15,22,18},pos;

    pos=busca(vector,7,15);
    if (pos >= 0)
        cout << "El elemento aparece en la posición " << pos << endl;
    else
        cout << "El elemento no se encuentra en el vector" << endl;
}

int busca(int v[], int tam, int elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;

}
```

En promedio, el **tiempo de ejecución** de busca depende **linealmente del tamaño** (tam) del array.

Del C al C++

- *Arrays*. Búsqueda binaria en *array* ordenado.

```
int buscaBinaria(int v[], int tam, int elem)
{
    int inf=0, sup=tam-1, medio ;

    while (sup > inf)
    {
        medio= (sup+inf)/2;
        if (v[medio] == elem)
            return medio;
        else if (v[medio] > elem)
            sup=medio-1;
        else
            inf = medio+1;
    }
    return -1;
}
```

- *Arrays*. Búsqueda binaria en array ordenado.

```
int buscaBinaria(int v[], int tam, int elem)
{
    int inf=0, sup=tam-1, medio ;

    while (sup > inf)
    {
        medio= (sup+inf)/2;
        if (v[medio] == elem)
            return medio;
        else if (v[medio] > elem)
            sup=medio-1;
        else
            inf = medio+1;
    }
    return -1;
}
```

En cada iteración el espacio de búsqueda se reduce a la mitad. En promedio, el **tiempo de ejecución de la búsqueda binaria** depende **logaritmicamente** ($\log_2 n$) del tamaño (n) del array.

Arrays. Métodos de ordenación. Ordenación por inserción.

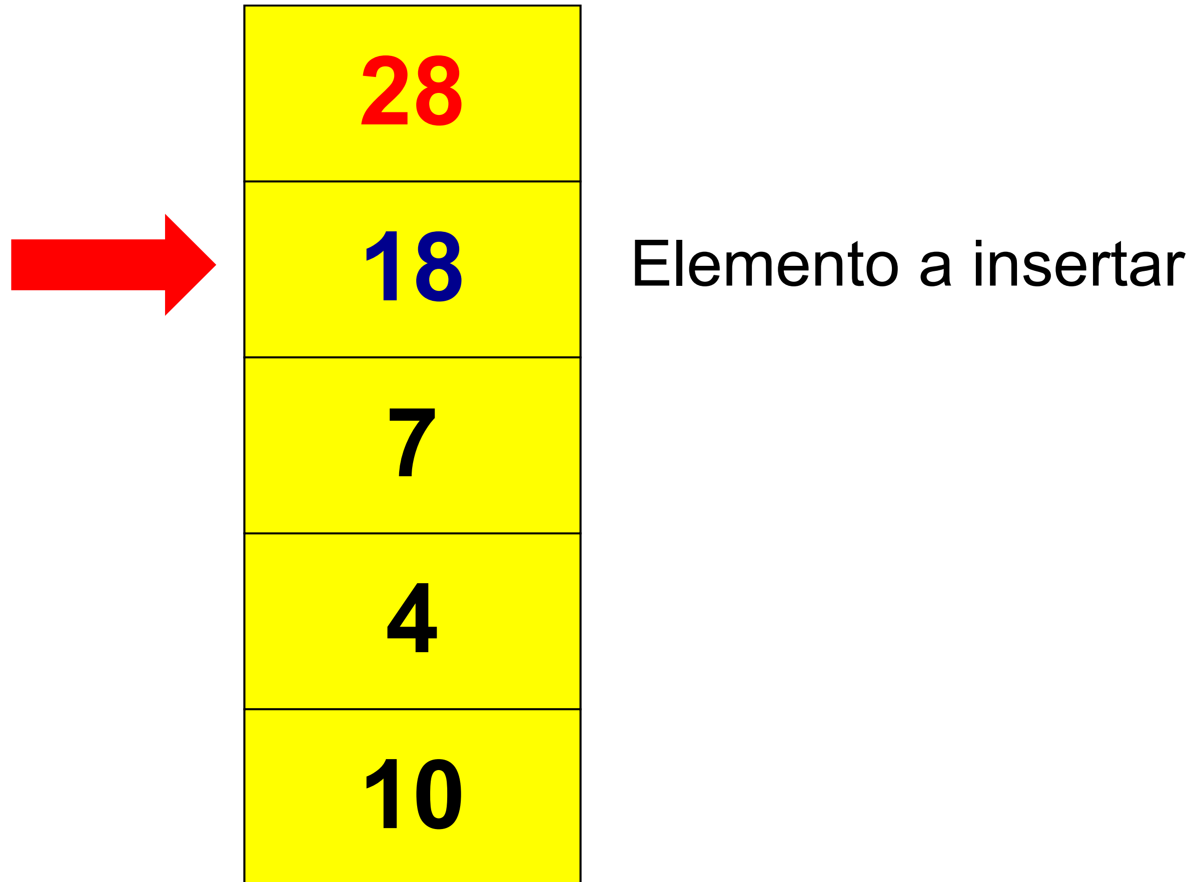
La idea básica es la siguiente:

1. Compare el segundo elemento con el primer elemento y si están desordenados intercámbielos.
2. Compare el tercer elemento con los dos anteriores e insértelo en la posición adecuada.
3. Compare el cuarto elemento con los tres anteriores e insértelo en la posición adecuada.
4. ....

Antes de empezar, puede considerarse que el primer elemento esta ordenado respecto a si mismo.

28
18
7
4
10

- *Arrays*. Métodos de ordenación. Ordenación por inserción.



- *Arrays*. Métodos de ordenación. Ordenación por inserción.

28
7
4
10

⟷ 18 Desplazamiento

Arrays. Métodos de ordenación. Ordenación por inserción.

18
28
7
4
10

Tras la primera iteración, tenemos ordenados entre si los dos primeros elementos.

En general, para un vector de n elementos, tras la iteración i estarán ordenados entre si los i+1 elementos del vector, siendo necesarias n-1 iteraciones.

Programar función que ordene según el método con el siguiente prototipo:

```
void OrdenaInsercion(int [], int);
```

- *Arrays*. Métodos de ordenación. Ordenación por inserción.

```
void ordenaInsercion(int v[], int tam)
{
    for (int i=1; i < tam; i++)
    {
        int aux=v[i];
        int j=i;
        while (j>0 && v[j-1] > aux)
        {
            v[j]=v[j-1];
            j--;
        }
        v[j]=aux;
    }
}
```

¿Cómo crees que aumenta el tiempo de ejecución de este método de ordenación en función del tamaño del array?

Arrays. Métodos de ordenación. Ordenación por inserción.

```
void ordenaInsercion(int v[], int tam)
{
    for (int i=1; i < tam; i++)
    {
        int aux=v[i];
        int j=i;
        while (j>0 && v[j-1] > aux)
        {
            v[j]=v[j-1];
            j--;
        }
        v[j]=aux;
    }
}
```

Como tiene un bucle dentro de otro, en promedio, el tiempo de ejecución depende aproximadamente del tamaño al cuadrado: n^2

Existen otros algoritmos como el **Quicksort** que tienen un tiempo de ejecución proporcional a $n*\log_2(n)$

Del C al C++

Complejidad algorítmica

- El estudio de la **complejidad de los algoritmos**: cómo crece el tiempo de ejecución o la utilización de memoria en función de algún índice del tamaño del problema (ej: **dimensión de los vectores y matrices, etc**), en una disciplina básica de la ciencia de la computación.
  - En promedio
  - En el mejor caso
  - En el peor caso
- Aunque no lo estudiaremos en profundidad, si es conveniente que los programadores se preocupen por estos temas.
- Está preocupación debe ser mayor en el caso de la **Informática Industrial**, porque aquí el cumplimiento de los plazos temporales es muy importante.

Del C al C++

Complejidad algorítmica

- En computación interesa el comportamiento límite, **asimptótico**, aproximado en la medida en que el tamaño del problema crece. Notación ***O grande*** (*Big O notation*).
- Ej: si f(n)

$$f(n) = 3n^2+2n+1 \Rightarrow f(n) = O(n^2)$$

- Y esto quiere decir que puedo encontrar una M de manera que:

$$f(n) < Mn^2 \qquad \text{para } n >= n_0$$

Del C al C++

Complejidad algorítmica

- En computación interesa el comportamiento límite, **asimptótico**, aproximado en la medida en que el tamaño del problema crece. Notación ***O grande*** (*Big O notation*).
- Ej: si f(n)

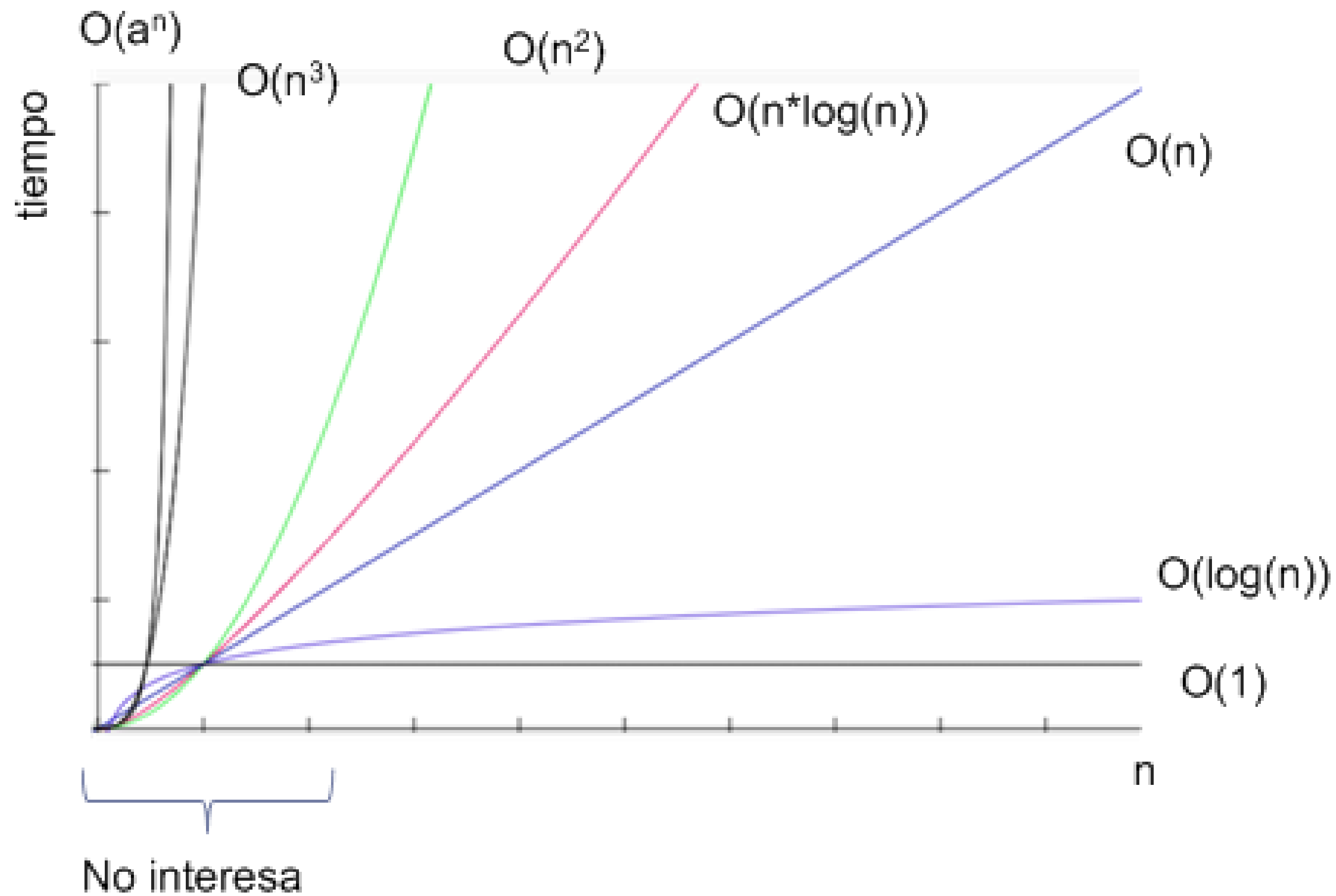
$$f(n) = 3n^2+2n+1 \Rightarrow f(n) = O(n^2)$$

- Y esto quiere decir que puedo encontrar una M de manera que:

$$f(n) < Mn^2 \quad \text{para } n >= n_0$$

Nos quedamos con n a la mayor potencia, ignorando las constantes que multiplican.

Complejidad algorítmica



Del C al C++

- *Complejidad algorítmica*

O(a^n)
O(n^3)
O(n^2)
O(n*log(n))
O(n)
O(log(n))
O(1)
tiempo
n
No interesa

Acceder al elemento i de un array

Del C al C++

Complejidad algorítmica

$O(a^n)$

$O(n^3)$

$O(n^2)$

$O(n*log(n))$

$O(n)$

$O(log(n))$

$O(1)$

tiempo

n

No interesa

Búsqueda binaria en una colección ordenada

Del C al C++

- *Complejidad algorítmica*

O(a^n)
O(n^3)
O(n^2)
O(n*log(n))
O(n)
tiempo

Búsqueda en una colección no ordenada

O(log(n))
O(1)
n

No interesa

Del C al C++

- *Complejidad algorítmica*

O(a^n)
O(n^3)
O(n^2)
O(n*log(n))
O(n)
O(log(n))
O(1)
tiempo
n
No interesa

Algoritmo de ordenación quicksort

Del C al C++

- *Complejidad algorítmica*

$O(a^n)$ $O(n^3)$ $O(n^2)$ $O(n*log(n))$ $O(n)$

tiempo

Algoritmo de ordenación de inserción

$O(log(n))$

$O(1)$

n

No interesa

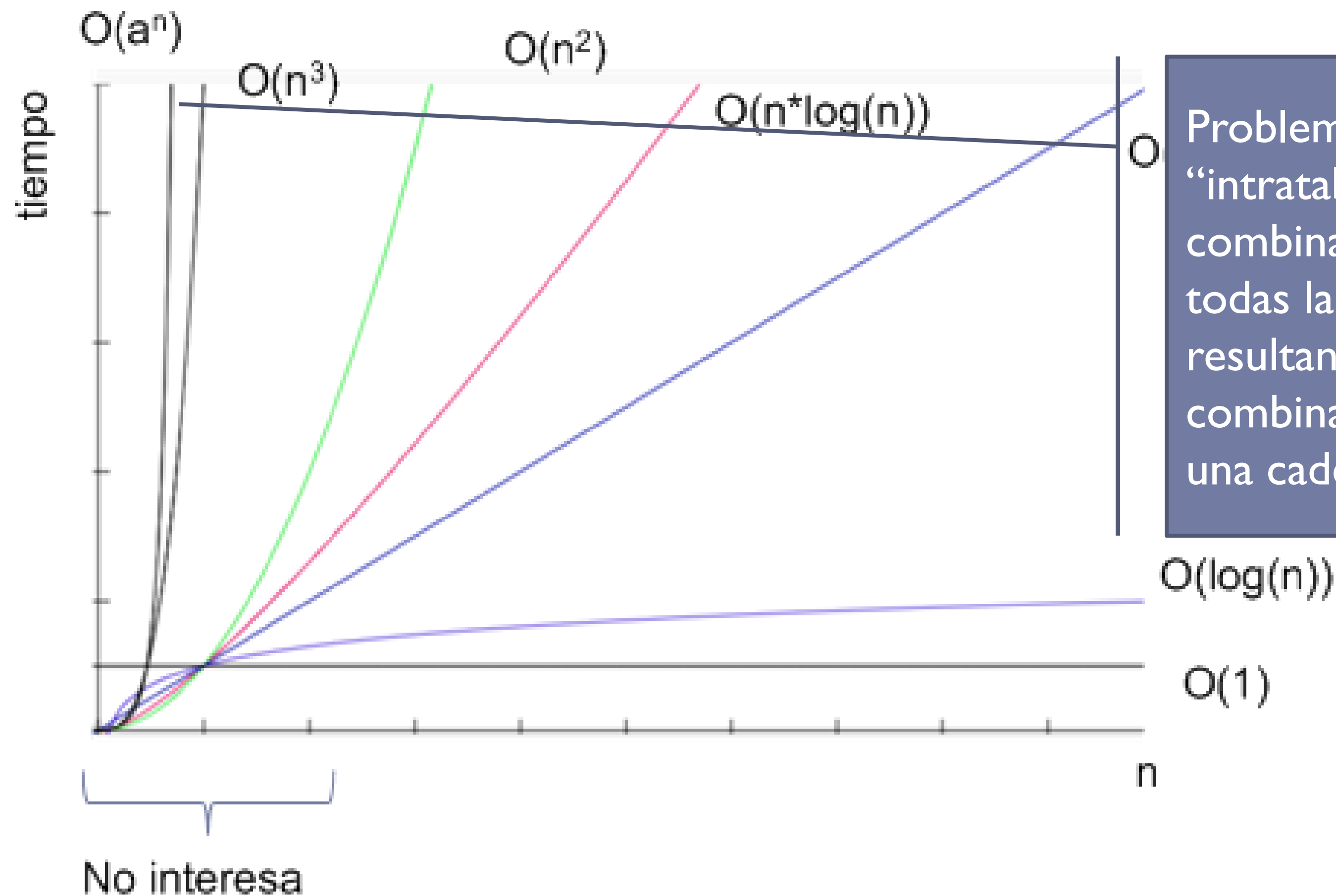
Complejidad algorítmica

O(a^n)
O(n^3)
O(n^2)
O(n*log(n))
O(n)
O(log(n))
O(1)
tiempo
n
No interesa

Algoritmo "ingenuo" de multiplicación de matrices

Del C al C++

- *Complejidad algorítmica*



Problemas básicamente "intratables", de tipo combinatorio. Ej: determinar todas las cadenas resultantes de la combinación de letras de una cadena dada.

Del C al C++

- *¿Y si queremos hacer búsqueda u ordenar no enteros?*

```
int busca(int [], int, int);
```

Del C al C++. Programación genérica.

- *¿Y si queremos hacer búsqueda u ordenar no enteros?*
- *¿Cómo se modifica el siguiente prototipo para buscar una cadena de cadena de caracteres?*

```
int busca(int [], int, int);
```

Del C al C++. Programación genérica.

- *¿Y si queremos hacer búsqueda u ordenar no enteros?*
- *¿Cómo se modifica el siguiente prototipo para buscar una cadena de cadena de caracteres?*

```
int busca(int [], int, int);

int busca(string [], int, string);
```

Ahora el *array* es del tipo de datos ***string***. El segundo argumento es el tamaño del *array* (sigue siendo **int**) y el tercero es el ***string*** a buscar. Sigue devolviendo el índice encontrado o -1.
Observad que ambos prototipos pueden convivir en el mismo programa aunque las funciones se llamen igual: el nombre de la función estaría **sobrecargado**.

Del C al C++. Programación genérica.

```
#include <iostream>
#include <string>

using namespace std;

int busca(string [], int, string);

int main()
{
    string vector[]={"uno","dos","encuentra","pos","diez","abc","c++"};
    int pos;

    pos=busca(vector,7,"encuentra");

    if (pos >= 0)
        cout << "El elemento aparece en la posición " << pos << endl;
    else
        cout << "El elemento no se encuentra en el vector" << endl;

    return 0;
}

int busca( string  v[], int tam, string elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;
}
```

Del C al C++. Programación genérica.

```
#include <iostream>
#include <string>

using namespace std;

int busca(string [], int, string);

int main()
{
    string vector[]={"uno","dos","encuentra","pos","diez","abc","c++"};
    int pos;

    pos=busca(vector,7,"encuentra");

    if (pos >= 0)
        cout << "El elemento aparece en la posición " << pos << endl;
    else
        cout << "El elemento no se encuentra en el vector" << endl;

    return 0;
}

int busca( string  v[], int tam, string elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;
}
```

La comparación funciona porque la clase ***string*** tiene sobrecargados este tipo de operadores.

Del C al C++. Programación genérica.

```
int busca( string  v[], int tam, string elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;
}

int busca(int v[], int tam, int elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;

}
```

Nótese que, gracias a las mencionadas caracterísitcas de C++, el código de ambas funciones es casi idéntico y que ambas funciones pueden convivir en el mismo programa y ser utilizadas de manera similar:

```
int cod[30];
string v[100];

…
busca(v,100,"cad");
busca(cod,30, 34);
```

y se llamaría a la función correspondiente en base a los argumentos reales.

Del C al C++. Programación genérica.

```
int busca( string  v[], int tam, string elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;
}

int busca(int v[], int tam, int elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;

}
```

Pero todavía el programador tiene que editar las dos (o más) versiones casi idénticas ...
Sería bueno que se puediera automatizar este proceso.

Esto es precisamente lo que hace la **programación genérica**.

Del C al C++. Programación genérica.

La programación genérica trabaja con el concepto de patrones (*templates*). Un patrón es un modelo para crear cosas similares.

En el ejemplo anterior, debemos comprender que lo único que cambia es el tipo de datos del vector y del valor a buscar.

```
#include <string>

using namespace std;

template <typename Tipo>
int busca(Tipo v[], int tam, Tipo  elem)
{
    for (int i=0; i < tam; i++)
    {
        if (v[i]== elem)
            return i;
    }
    return -1;
}
```

Definimos el patrón de la función que deberá ser generada automáticamente cuando el programado llame a la función busca con un tipo (Tipo) de datos concreto.

Tenemos que aislar el tipo de dato que cambia, de una variante a otra.

Los *templates* tienen que definirse al inicio del programa, de hecho es preferible hacerlo en ficheros de cabecera.

Del C al C++. Programación genérica.

A partir del tipo de dato con el que se llame a la función genérica, el sistema creara la versión de la función sobrecargada necesaria, de manera transparente al programador.

```
int main()
{
    int vec_int[]={1,4,10,3,15,22,18};
    string vec_string[]={"uno", "tres", "cuatro"};
    char vec_char[]={'a','c','b'};

    cout << "En vec. enteros el elemento 10 aparece en " << busca(vec_int,7,10) << endl;

    cout << "En vec. de strings el elemento tres aparece en " << busca(vec_string,3,(string)"tres") << endl;

    cout << "En vec. chars el elemento b aparece en " << busca(vec_char,3,'b') << endl;
}
```

Del C al C++. Programación generica.

- Ejercicio: Cree una versión genérica de la función que intercambia dos variables de manera que funcione con cualquier tipo de datos.

Biblioteca STL

- C ++ viene con una biblioteca integrada por clases reutilizables para construir programas. Esta biblioteca se llama biblioteca de plantillas estándar C++ (STL).
- La biblioteca STL es un conjunto de clases que proporcionan contenedores, algoritmos e iteradores.
- Se basan en el concepto de programación genérica.
- Existen (entre otras muchas) elementos contenedores que son clases genéricas que encapsulan el comportamiento de los arrays estáticos y dinámicos (array<> y vector <>).

Contenedores Secuenciales. La clase Array

- **La clase array** son contenedores secuenciales de **tamaño fijo**: contienen un número específico de elementos ordenados en una secuencia lineal estricta.
- Es tan eficiente en términos de tamaño de almacenamiento como un *array* declarado tradicionalmente.
- Esta clase simplemente añade una capa de funciones miembro.
- A diferencia de los otros contenedores estándar, arrays tienen un **tamaño fijo y no pueden expandirse o contraerse dinámicamente** (a continuación veremos el vector para un contenedor similar que se puede ampliar).

Contenedores Secuenciales. La clase Array

```
#include <iostream>
#include <array>

using namespace std;

template <typename T, std::size_t tam>
void mul(array<T,tam> &a)
{
    for (auto &elem: a)
        elem*=2;
}

int main()
{
    array<int,5> v {1, 2 ,3, 4, 5};

    for (auto i=0; i!=v.size(); i++)
        cout << v[i];

    cout << endl;
    mul(v);

    for (auto elem: v)
        cout << elem;

    cout << endl;

}
```

Función mul genérica:

- Paso por referencia del array
- Uso del for de rango con **auto &** porque modificará los elementos.

Hay que definir el tipo de elementos contenido y el tamaño. Este ejemplo usa la nueva inicialización universal

Observe:

- Uso de auto cuando el tipo puede ser "deducido" por el compilador
- Método size() devuelve tamaño del array

for de rango con **auto**

Contenedores Secuenciales. La clase Vector

- **La clase** vector de la STL es un *array* dinámico **capaz de crecer** según sea necesario para contener sus elementos.
- Este crecimiento se realizará mediante reserva dinámica de memoria pero la clase vector encapsula, "oculta" estos detalles.
- La clase vector **permite acceso aleatorio** a sus elementos a través del operador [], y la inserción y extracción de elementos del extremo del vector es rápida.

Iteradores

- Un **iterador** es un objeto que puede recorrer (iterar sobre) una clase de contenedor sin que el usuario tenga que conocer cómo se implementa el contenedor.
- Un **iterador** es como un puntero a un elemento dado en el contenedor, con un conjunto de operadores sobrecargados para proporcionar un conjunto de funciones bien definidas.
- En el caso del vector constituye una alternativa a []

Contenedores Secuenciales. La clase Array

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<double> v {1.1, 2.2 ,3.2, 4.4, 5.5};

    for (auto i=0; i!=v.size(); i++)
        cout << v[i] << " ";
    cout << endl;

    vector<double>::iterator it=v.begin();

    while (it!= v.end())
    {
        *it=*it + 2;
        it++;
    }
    for (int i=0; i != 10; i++)
        v.push_back((double) i);

    for (auto elem: v)
        cout << elem << " ";
    cout << endl;
}
```

Vector <> es inicializado.

Se puede recorrer como array tradicional. Método size tiene el tamaño actual.

Observe uso de iteradores como generalización genérica de los punteros:

- Una definición equivalente:

auto it= v.begin()

El vector puede crecer de tamaño.

Algoritmos

- Existen muchos algoritmos capaces de trabajar con las clases contenedoras como vector.
- Debe incluirse el fichero de cabecera <algorithm>
- Constituye código muy optimizado. Se recomienda antes de intentar una solución propia, comprobar si existe un algoritmo genérico que realice la misma función.
- El funcionamiento de los algoritmos puede ser "personalizado" utilizando **funciones** hechas a la medida, o **functores** (clases que sobrecargan el **operador ()**), o las nuevas funciones mal llamadas anónimas (**lambdas**).

Algoritmos

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib> //para rand
#include <chrono>

using namespace std;

void imprime(vector<int> &v)
{
    for (auto elem: v)
        cout << elem << " ";
    cout << endl;
}
int aleatorio()
{
    return rand()%100;
}
int main()
{
    vector<int> v;

    srand(chrono::system_clock::to_time_t(chrono::system_clock::now()));

    for (auto i=0; i < 10; i++)
        v.push_back(aleatorio());
    imprime(v);
    generate(v.begin(), v.end(), aleatorio);
    imprime(v);
    sort(v.begin(),v.end());
    imprime(v);
}
```

Función que devuelve un numero aleatorio entre 0 y 100

Proporcionamos "seed" al generador de números aleatorios en función del tiempo actual. Biblioteca <chrono>

```
11 75 45 10 34 41 19 54 24 78
67 3 54 41 70 59 52 29 18 70
3 18 29 41 52 54 59 67 70 70
```

Ej. salida por consola

Observe uso algoritmo **generate** (que usa la función aleatorio) para rellenar vector y del algoritmo **sort** para ordenarlo

Algoritmos

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>

using namespace std;
template<typename T>
void imprime(vector<T> &v)
{
    for (auto elem: v)
        cout << elem << " ";
    cout << endl;
}
class GenRango
{
    double actual;
    double incremento;
public:
    explicit GenRango(double mn, double inc):actual {mn},incremento {inc} {}
    double operator() ()
    {
        return actual+=incremento;
    }
};
int main()
{
    vector<double> v(10);
    GenRango g(100, 5);

    generate(v.begin(),v.end(),g);
    imprime(v);
    sort(v.begin(),v.end(),[](double r,double l) {return r>l; });
    imprime(v);
}
```

Functor. Clase que sobrecarga el operador de llamada a funciÓn

```
105 110 115 120 125 130 135 140 145 150
150 145 140 135 130 125 120 115 110 105
```

Ej. salida por consola

Algoritmo generate con functor.

Sort con función lambda para ordenar de mayor a menor.

Algoritmos

Investigar eficiencia computacional del algoritmo **sort** utilizando como ejemplo el siguiente programa.

Investigar otros algoritmos disponibles en biblioteca estándar.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib> //para rand
#include <chrono>

using namespace std;

int aleatorio()
{
    return rand();
}
int main()
{
    vector<int> v(100000);

    srand(chrono::system_clock::to_time_t(chrono::system_clock::now()));

    for_each(v.begin(),v.end(),[](int &elem){elem=aleatorio();});

    auto t0=chrono::high_resolution_clock::now();
    sort(v.begin(),v.end());
    auto t1=chrono::high_resolution_clock::now();

    auto lapso= chrono::duration_cast<chrono::microseconds> (t1-t0).count();

    cout << lapso << " microsegundos " << endl;
}
```

Informática Industrial

Práctica 5. Listas enlazadas. Clase lista en STL. Adaptadores en STL: priority queue

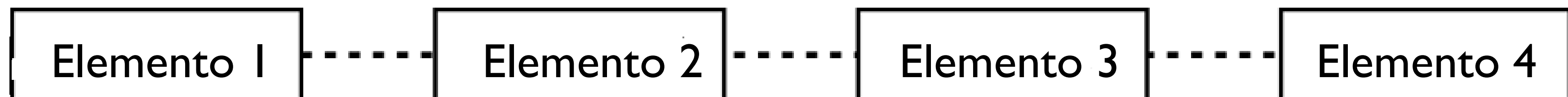
Del C al C++

- La lista enlazada. Comparación con el *array*.
- Otras colecciones de datos.
- Las listas enlazadas en la biblioteca STL.

Listas

Definición

- Son secuencias de elementos del mismo tipo
- Existe un elemento inicial y otro final
- Cada elemento tiene un predecesor y un sucesor, salvo el inicial que no tiene predecesor y el final que no tiene sucesor
- Los *arrays* son un ejemplo de listas
- Las listas enlazadas brinda una forma diferente de programar las listas



Listas simplemente enlazadas

Deben utilizarse cuando el tamaño de las listas es impredecible y varía dinámicamente con la vida del programa

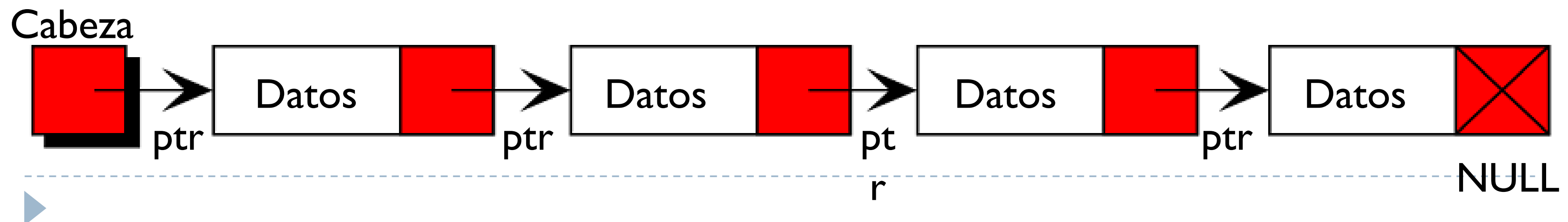
En una lista enlazada los elementos son independientes unos de otros, y en general están almacenados en posiciones no contiguas

Cada elemento (nodo) consta de dos partes diferenciadas:

- La zona de datos, que puede corresponder a cualquier combinación de tipos de datos válidos en C (incluidas otras estructuras)
- Un enlace (puntero), que almacena la dirección en memoria del siguiente nodo de la lista

Al ser los elementos independientes, pueden realizarse operaciones de borrado e inserción con facilidad y en cualquier parte de la lista

- Aparte de reservar memoria (inserción) o liberarla (borrado), será necesario actualizar los punteros de los elementos involucrados

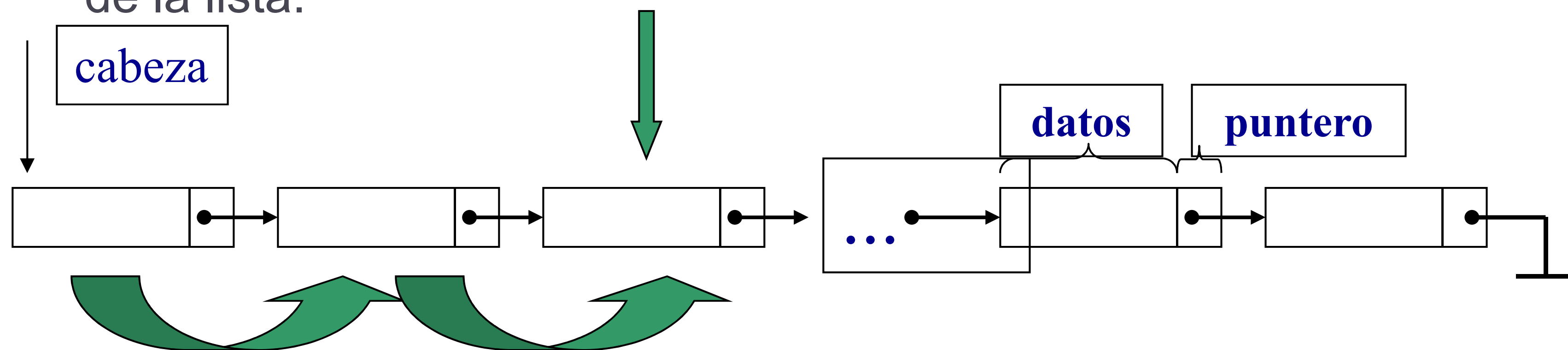


Listas simplemente enlazadas

Es una organización más costosa en memoria (además de los datos propiamente dichos hay que almacenar el puntero).

El acceso a los elementos de datos se **realiza de forma secuencial y en un único sentido** (es una operación **O(n)** en promedio), lo cual es más lento (no es posible acceso aleatorio) que el **O(1)** de los arrays.

Ej: Si se quiere llegar al **elemento 3** de las lista, hay que recorrer primero los 2 primeros desde el puntero inicial marcado por la **cabeza** de la lista.



No es necesario tener otra variable que recoja el tamaño de la lista. Se sabe que termina cuando el elemento siguiente es **NULL**.

Si ya estamos en la posición adecuada, la inserción y el borrado es **O(1)** a diferencia del **O(n)** del *array*.

Listas simplemente enlazadas

Veremos un ejemplo de lista enlazada simple para comprender la naturaleza de los mecanismos básicos. No haremos uso de todos los elementos O.O. de C++. Más adelante utilizaremos los contenedores genéricos mucho más potentes.

```
#include <iostream>

using namespace std;

struct Nodo
{
    int dato;
    Nodo *sig;
};
/*Inserta manteniendo orden*/
void Insertar(Nodo* &cab,int dato);
int  Borrar(Nodo * &cab,int dato);
void Imprimir(Nodo * cab);
enum Menu {salir=0,borr=1,ins=2,imp=3};
```

Se define una estructura (o clase) con el campo(s) que almacenará los datos y el que apunta al siguiente elemento en la lista.

Cuando la función puede modificar el puntero a la cabecera de la lista, se pasa una referencia a dicho puntero.

Listas simplemente enlazadas

```
int main()
{
    Nodo *cabeza=NULL;
    int opcion;
    int dat;

    do
    {
        cout << "Diga opcion\n0- Salir\n1-Borrar\n2-Insertar\n3-Imprimir\n\nOpcion: " << endl;
        cin >> opcion;
        switch (opcion)
        {
            case borr:
                        cout << "Diga entero a borrar: "<<endl;
                        cin >> dat;
                        if (Borrar(cabeza, dat)==0)
                            cerr << "Dato no existía" <<endl;
                        break;
            case ins:
                        cout << "Diga entero a insertar: "<<endl;
                        cin >> dat;
                        Insertar(cabeza,dat);
                        break;
            case imp:
                        Imprimir(cabeza);
                        break;
        }
    }while (opcion != salir);
    return 0;
}
```

Listas simplemente enlazadas

```
void Imprimir(Nodo * cab)
{
    while (cab!=NULL)
    {
        cout << "Elemento: " << cab->dato << endl;
        cab=cab->sig;
    }
}
```

Listas simplemente enlazadas

```
/*Inserta nodos en orden de menor a mayor*/
void Insertar(Nodo* &cab,int dato)
{
    Nodo *nuevo;
    Nodo *actual;
    Nodo *anterior;

    nuevo = new Nodo;
    nuevo->dato=dato;
    actual=cab;
    anterior=NULL;
    if (cab==NULL)
    {
        cab=nuevo;
        nuevo->sig=NULL;
    }
```

Se reserva dinámicamente espacio para el nuevo nodo.

Si la lista está vacía, se hace que la cabecera apunte al nuevo elemento y que el campo siguiente de esta contenga NULL. Notad que cab ha sido pasado por referencia.

(... cont)

Listas simplemente enlazadas

```
else{
    while (actual!=NULL && actual->dato < dato)
    {
        anterior=actual;
        actual=actual->sig;
    }
    if (anterior==NULL)
    {
        cab=nuevo;
        nuevo->sig=actual;
    }
    else {
        anterior->sig=nuevo;
    }
    if (actual==NULL)
    {
        nuevo->sig=NULL;

    }else{
        nuevo->sig=actual;
    }
  }
}
```

Si la lista no está vacía, "navegamos" mientras no lleguemos al final y mientras no hallamos llegado al lugar dónde corresponde insertar. El puntero anterior apunta al elemento previo o a NULL si es el primero.

Notad que modificamos **actual** y no **cab** ¿Por qué es importante?

Si se debe borrar el primero, hay que modificar cab.

Listas simplemente enlazadas

```
/*Borra elemento*/
int  Borrar(Nodo * &cab,int dato)
{
    Nodo *actual;
    Nodo *anterior;


    if (cab==NULL)
        return 0;

    if (cab->dato == dato)
    {
        actual = cab;
        cab=cab->sig;
        delete actual;
        return 1;
    }
    anterior=cab;
    actual=cab->sig;
    while (actual != NULL && actual->dato < dato)
    {
        anterior=actual;
        actual=actual->sig;

    }
```

Si la lista está vacía, no hay nada que hacer.

Si hay que borra el primer elemento, se debe modificar la cabecera.

En caso contrario, se "navega" hasta encontrar el elemento o llegar al final.

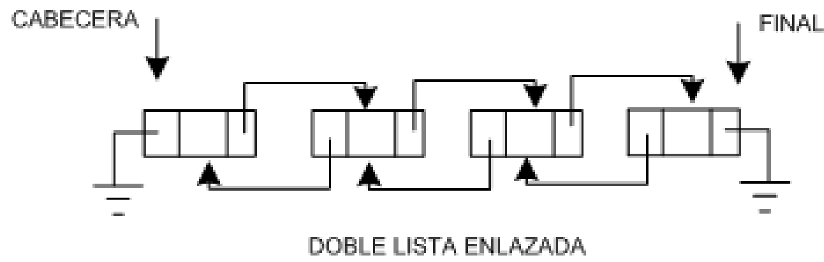
(... cont)

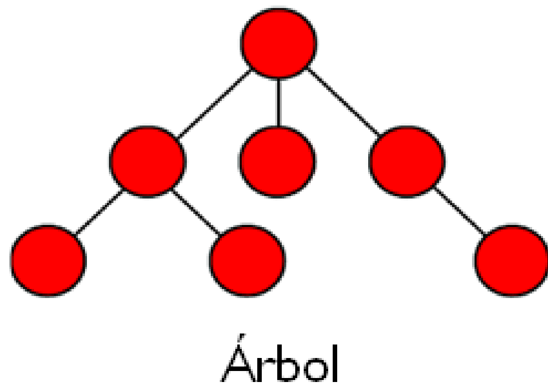
Listas simplemente enlazadas

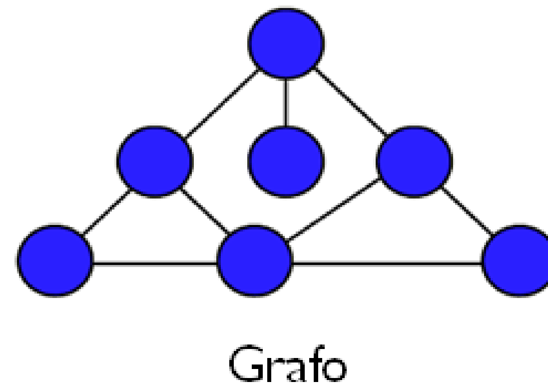
```
    if (actual==NULL)
        return 0;
    else{
        if (actual->dato == dato)
        {
            anterior->sig=actual->sig;
            delete actual;
            return 1;
        }else
            return 0;
    }
}
```

Estructura de otras colecciones de datos

Otras estructuras de datos.







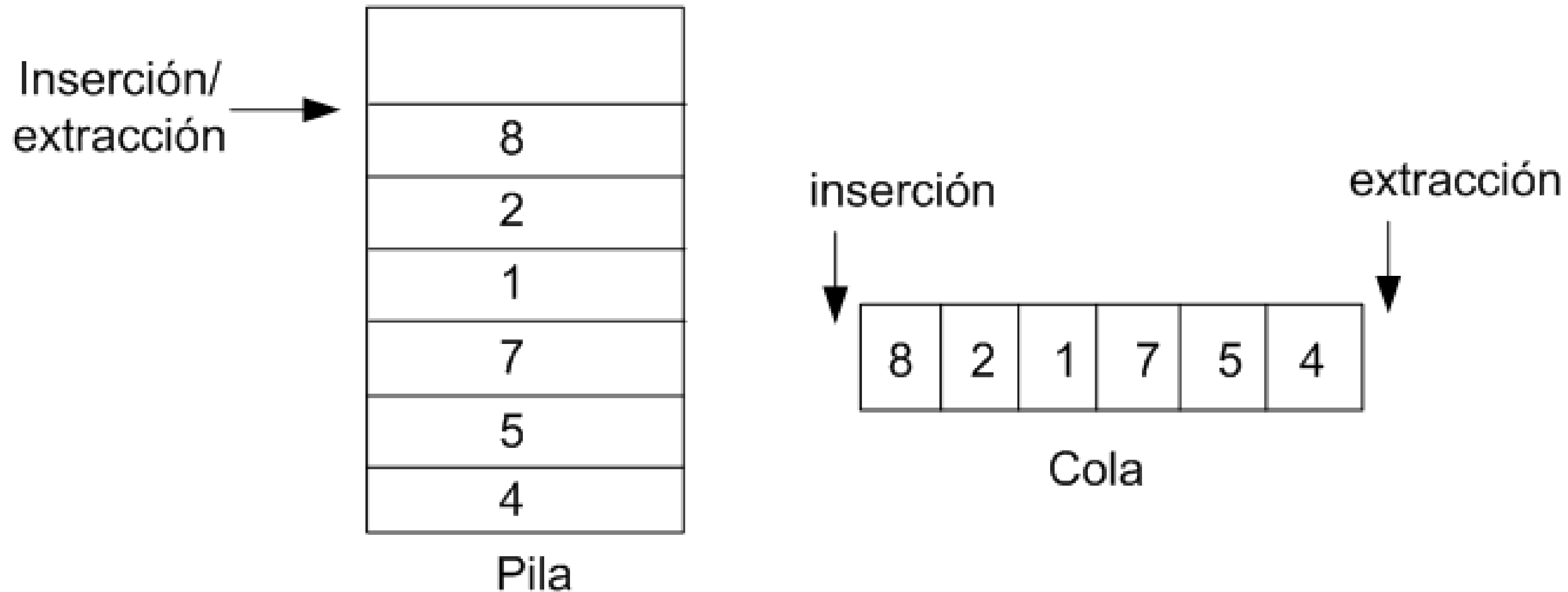
Estructura de otras colecciones de datos

Son muy útiles determinadas **restricciones** de como usar elementos tales como las **listas** y los **arrays**.

Ejemplos: las estructuras LIFO (*last in first out*) o **pilas** (*stacks*) sólo permiten introducir y sacar elementos por el mismo extremo.

Por el contrario, en las **colas** (*queues*) o estructura FIFO (*first in first out*) se introducen elementos por un extremos y se extraen por el otro.

Estructura de otras colecciones de datos



Tanto las pilas como las colas pueden ser programadas utilizando los arrays o las listas enlazadas como estructura base.

Ejercicio: Programad una pila utilizando una lista simplemente enlazada.

Contenedores Secuenciales. La clase **deque**

- Deque es un acrónimo de *double ended queue* y es una clase que implementa un *array* dinámico **capaz de crecer o encoger por ambos extremos**.
- Se puede **acceder directamente** a los elementos individuales.
- El **almacenamiento se gestiona automáticamente** en la expansión y contracción del contenedor según sea necesario. Por lo tanto, proporcionan una funcionalidad similar a los vectores , pero con la inserción y supresión de eficiente elementos también en el principio de la secuencia, y no sólo en su extremo.
- Pero, a diferencia de los **vectores**, las **deques** no garantizan el almacenamiento contiguo de todos sus elementos.
- Vectores y deques proporcionan una interfaz muy similar y se pueden utilizar indistintamente pero internamente operan de maneras muy diferente: mientras que los vectores utilizan un único *array* que necesita ser reasignado de vez en cuando para el crecimiento, los elementos de un **deque** pueden estar dispersos en diferentes partes de la memoria.
- **deques** son un poco más complejas interiormente que los vectores , pero esto les permite crecer de manera más eficiente, especialmente con secuencias muy largas.

Contenedores Secuenc. La clase **forward_list**

- La clase forward list es un contenedor que implementa una lista simplemente enlazada.
- La ventaja de las listas es que la inserción y borrado de elementos una vez localizado donde insertar/borrar es muy rápido (y en tiempo constante).
- El principal inconveniente de **forward_list** en comparación con los **vectores** es que **carecen de acceso directo** a los elementos por su posición.
- Generalmente se utilizan **iteradores** para recorrer la lista.

Contenedores Secuenciales. La clase **deque**

```cpp
#include <iostream>
#include <deque>
int main()
{
    using namespace std;

    deque<int> miLista;
    for (int nCount=0; nCount < 3; nCount++)
    {
        miLista.push_back(nCount); // insert at end of array
        miLista.push_front(10 - nCount); // insert at front of array
    }

    for (int nIndex=0; nIndex < miLista.size(); nIndex++)
        cout << miLista[nIndex] << " ";

    cout << endl;
}
```

Contenedores Secuenc. La clase **list**

- Una **lista** list es un contenedor que implementa una lista doblemente enlazada donde cada elemento en el contenedor contiene punteros que apuntan a los elementos siguiente y anterior en la lista.
- Las listas doblemente enlazadas proporcionan acceso al inicio y al final de la lista. Permiten el recorrido de atrás a adelante pero **no hay acceso aleatorio**.
- Generalmente se utilizan **iteradores** para recorrer la lista.

Iteradores

```
#include <iostream>
#include <list>
int main()
{
    using namespace std;

    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);

    list<int>::iterator it; // declare an iterator
    it = li.begin(); // assign it to the start of the list
    while (it != li.end()) // while it hasn't reach the end
    {
        cout << *it << " "; // print the value of the element it points to
        it++; // and iterate to the next element
    }
}
```

listas

```
#include <iostream>
#include <list>
using namespace std;

template <typename T>
void imprime(T v)
{
    for (auto e: v)
        cout<< e << " ";
    cout << endl;
}
int main()
{
    list<int> li {3,1,2};

    li.sort();

    li.push_front(4);
    li.push_back(10);

    imprime (li);
    for (auto it=li.begin(); it != li.end();it++)
        if (*it == 3)
        {
            li.insert(it,20);
            break;
        }
    imprime(li);
    li.remove(20);
    imprime(li);
}
```

```
4 1 2 3 10
4 1 2 20 3 10
4 1 2 3 10
```

Método sort de lista.

Métodos push_front y push_back para introducir valores al inicio y final de la lista.

Con iterador buscamos elemento 3 para insertar 20 frente a él.

Borramos elemento de valor 20

Salida por pantalla

Adaptadores.

- Los adaptadores stack, queue, priority queue están basados en los contendores básicos previamente discutidos, y básicamente limitan el funcionamiento de los contenedores subyacentes para ofrecer la funcionalidad esperada. Ej:
  - **stack** (por defecto se basa en lista) implementa LIFO y admite métodos como push y pop
  - **queue** (por defecto basado en deque) implementa FIFO y admite push_back , pop_front.
  - **priority_queue** (por defecto basado en vector) permite introducir elementos (push) pero el que se saca (pop) es siempre el mayor de acuerdo a algún criterio.

Priority queue

```cpp
#include <iostream>
#include <queue>

using namespace std;


struct mensaje
{
    string msg;
    int prioridad;
    bool operator <(const mensaje &rhs) const
      {return prioridad < rhs.prioridad;}
};

int main()
{

    priority_queue <mensaje> pq;


    pq.emplace(mensaje{"uno",5});
    pq.emplace(mensaje{"dos",10});
    pq.emplace(mensaje{"tres",0});

    while (!pq.empty())
    {
        mensaje m = pq.top();
        cout << m.msg << endl;
        pq.pop();
    }
}
```

Mensaje para almacenar en la cola con prioridades. Observe método de comparación

Definimos nuesta cola con prioridades

Colocamos varios elementos com emplace.

Obtenemos el elemento en orden de prioridad y posteriormente lo sacamos de la cola

```
dos
uno
tres
```

Salida por pantalla