

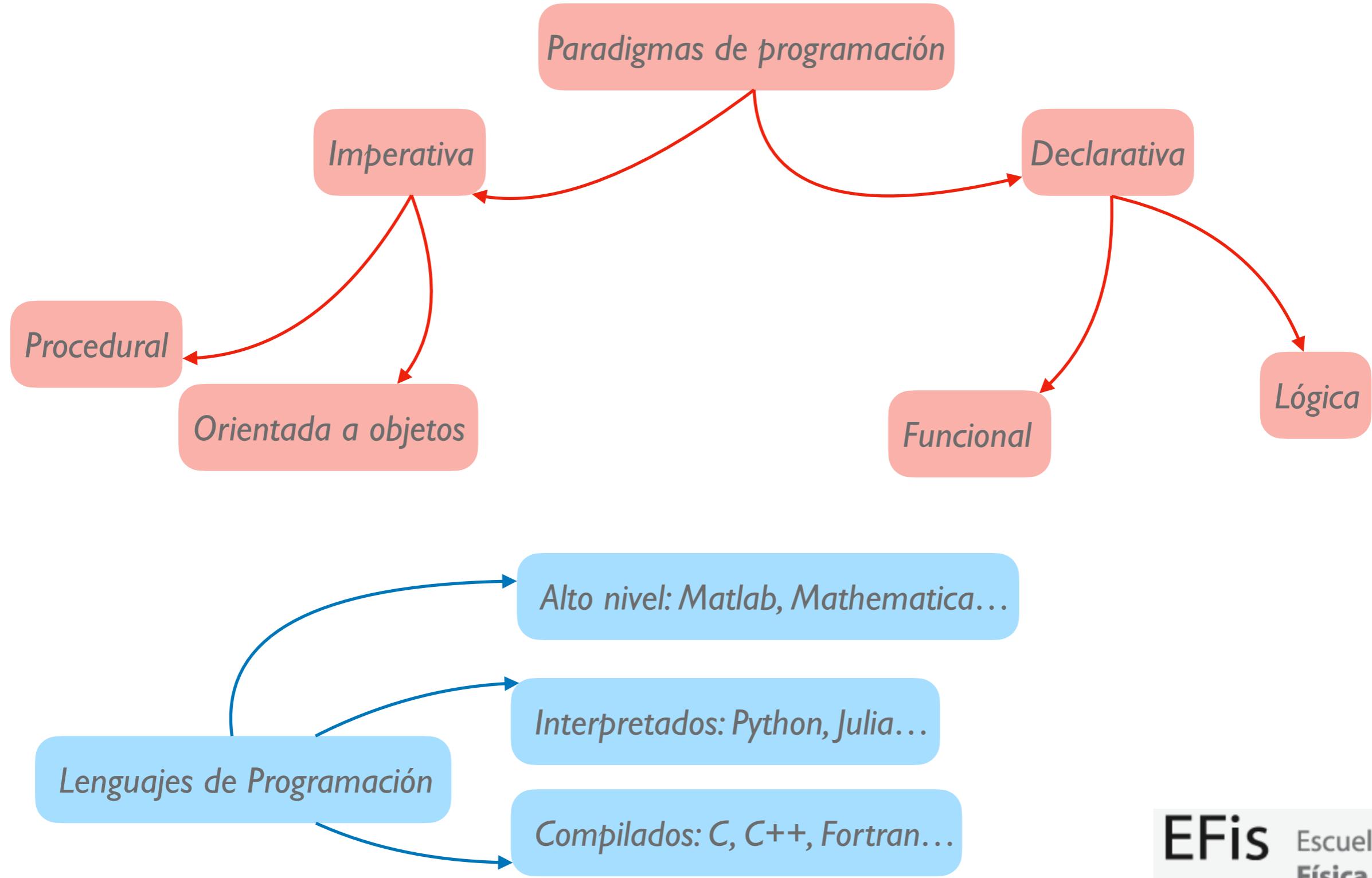


UNIVERSIDAD DE COSTA RICA

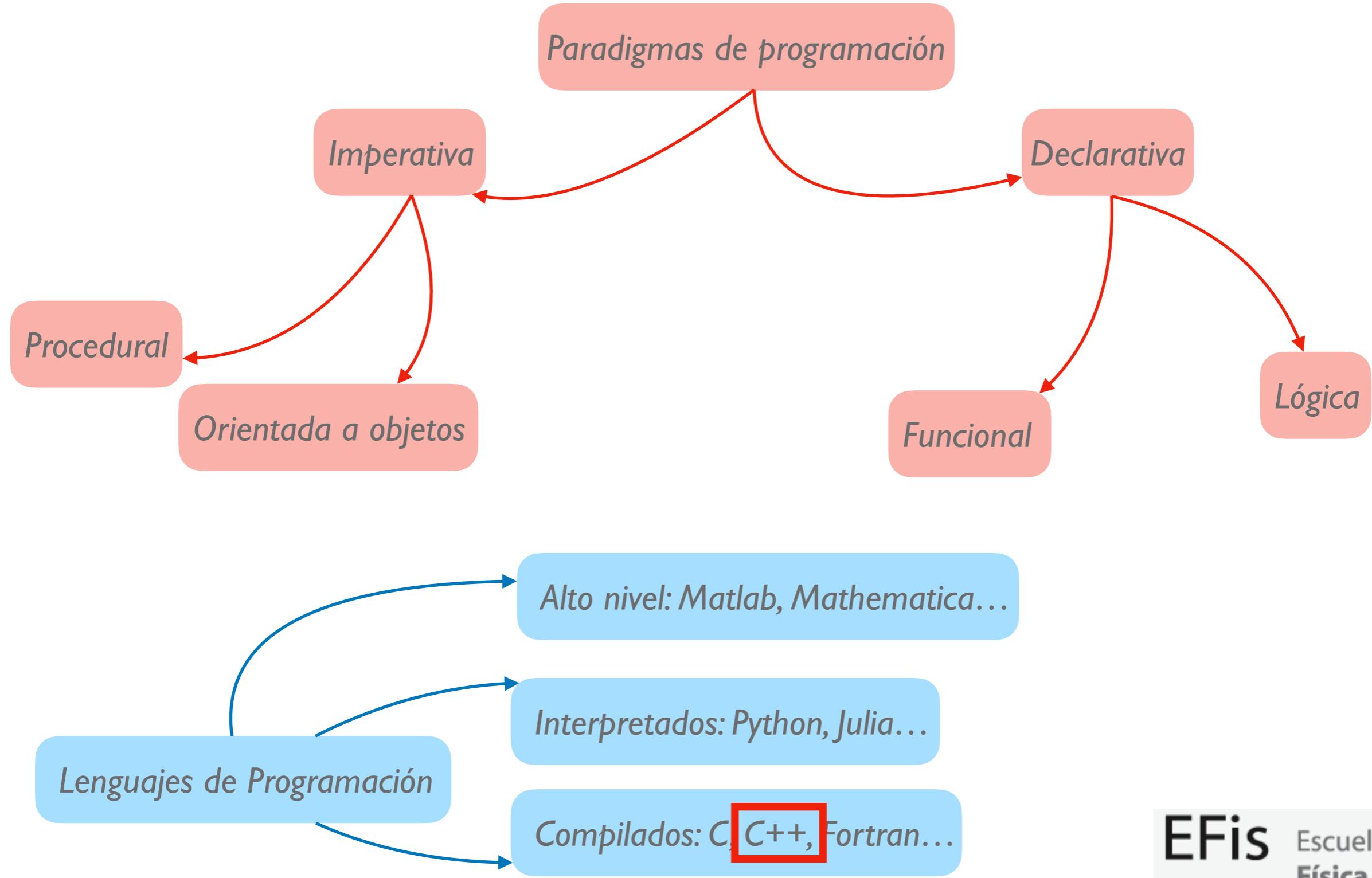
PROGRAMACIÓN COMPILADA: C++

Prof. Marlon Brenes y Prof. Federico Muñoz
Escuela de Física, Universidad de Costa Rica

Programación Intermedia: Guía de Curso



Programación Intermedia: Guía de Curso

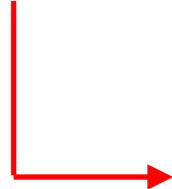


Lenguajes compilados

- Un programa llamado *compilador* toma las instrucciones y las traduce a lenguaje de máquina
- El resultado es una aplicación que realiza un objetivo mediante código escrito bajo cierto paradigma computacional
- Razones para usar lenguajes compilados:
 - Eficiencia: Las operaciones se ejecutan directamente en procesador, no hay etapa de interpretación
 - Interoperabilidad: Los lenguajes compilados e interpretados se pueden entrelazar
 - Control: Los lenguajes compilados permiten control directo sobre los recursos de la máquina

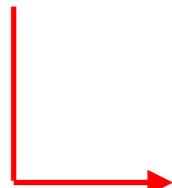
Ventajas y desventajas

- Ventajas



- Velocidad
- Control de memoria
- Mejor soporte de procesamiento en paralelo

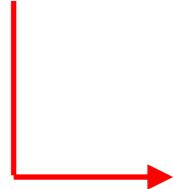
- Desventajas



- Dificultad
- Dependencia del hardware
- Ciclos de desarrollo más largos y más complicados

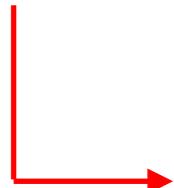
C++

- C++ es un lenguaje de propósito general de paradigma imperativo



- Programación orientada a objetos y genérica
- Control de memoria de bajo nivel
- Creado por Bjarne Stroustrup (AT&T Bell Labs 1983)

- C++ incluye toda la funcionalidad de C y adicionalmente:



- Objetos
- Excepciones
- Polimorfismo
- Confirmación de tipos a nivel de *run-time*

Variables y declaraciones

- Variables

```
bool    // Boolean, possible values are true and false
char    // character, for example, 'a', 'z', and '9'
int     // integer, for example, 1, 42, and 1216
double  // double-precision floating-point number, for example, 3.14 and 299793.0
```

- Operaciones

+	// plus, both unary and binary	==	// equal
-	// minus, both unary and binary	!=	// not equal
*	// multiply	<	// less than
/	// divide	>	// greater than
%	// remainder	<=	// less than or equal
		>=	// greater than or equal

Estructura básica

Archivo header

Función de un argumento sin valor de retorno

```
1 #include <iostream>
2
3 void func(std::string name){
4
5     std::cout << "Hello, " << name << "!" << std::endl;
6 }
7
8 int main(){
9
10    std::string my_name = "Marlon";
11
12    func(my_name);
13
14    return 0;
15 }
```

Impresión a std::out

Función principal

Declaración de variable

Llamado a la función

• hello.cpp

Compilación y ejecución

```
[mbrenes0@hxckid:Semana_02$ g++ hello.cpp -o hello.x
[mbrenes0@hxckid:Semana_02$ ls hello.x
hello.x
[mbrenes0@hxckid:Semana_02$ ./hello.x
Hello, Marlon!
mbrenes0@hxckid:Semana_02$ ]
```

Inicialización y regiones de variables (scopes)

```
1 #include <iostream>
2
3 int main(){
4
5     int i = 5;
6     int a = 3;
7
8     std::cout << "i before loop: " << i <<
9         " a before loop: " << a << std::endl;
10
11    for(int i = 0; i < 10; ++i){
12        {
13            int a = 7;
14            int b = 3;
15            std::cout << "i inside loop: " << i << " a inside loop: " << a << std::endl;
16        }
17    }
18    std::cout << "i after loop: " << i << " a after loop: " << a << std::endl;
19    // std::cout << b << std::endl; <--- b es invisible en este scope
20
21    return 0;
22 }
```

Diferentes scopes se refieren a variables diferentes, aunque tengan el mismo nombre

Los scopes se denominan mediante {}

- mid-code_ini.cpp

Namespaces

- Los *namespaces* son grupos de variables, objetos y funciones
- Su propósito es **evitar colisiones de nombramiento**
- Son muy útiles para organizar código que incluye múltiples bibliotecas

Namespaces

```
1 #include <iostream>
2
3 namespace A{
4     int var = 3;
5     void greet(){
6         std::cout << "Hello!" << std::endl;
7     };
8 }
9
10 namespace B{
11     int var = 4;
12     void greet(){
13         std::cout << "Hi!" << std::endl;
14     };
15 }
16
17 using namespace A;
18
19 int main(){
20
21     std::cout << "a = " << A::var << std::endl;
22     A::greet();
23     std::cout << "a = " << var << std::endl;
24     greet();
25     std::cout << "a = " << B::var << std::endl;
26     B::greet();
27
28     return 0;
29 }
```

Esta directriz hace todos los símbolos en A disponibles sin la necesidad de usar el operador de resolución del scope '::'. 'Poblar' el namespace de manera global es usualmente una mala práctica!!

Equivalentes dada la directriz
“using namespace A”

- namespace.cpp

Standard input

std::cin permite input de la terminal (stdin)

```
1 #include <iostream>
2
3 int main(){
4
5     std::cout << "What's your name?" << std::endl;
6
7     std::string name;
8     std::cin >> name;
9
10    std::cout << "Hello " << name << "!" << std::endl;
11
12    return 0;
13 }
```

- hello_again.cpp

- Una forma más concisa de escribir este programa es mediante la directriz global `using namespace std`
- Usualmente es mejor no poblar el namespace de manera global

Punteros y referencias de memoria

- Punteros



- Un puntero es un tipo básico en C/C++ que almacena un valor: una dirección lógica en memoria
- Los punteros también tienen tipos (int, float, double, char,...) y su tipo depende de a cual objeto apuntan
- Se declaran con el operador *

```
int *p1;           float *num1;           double *num2;
```

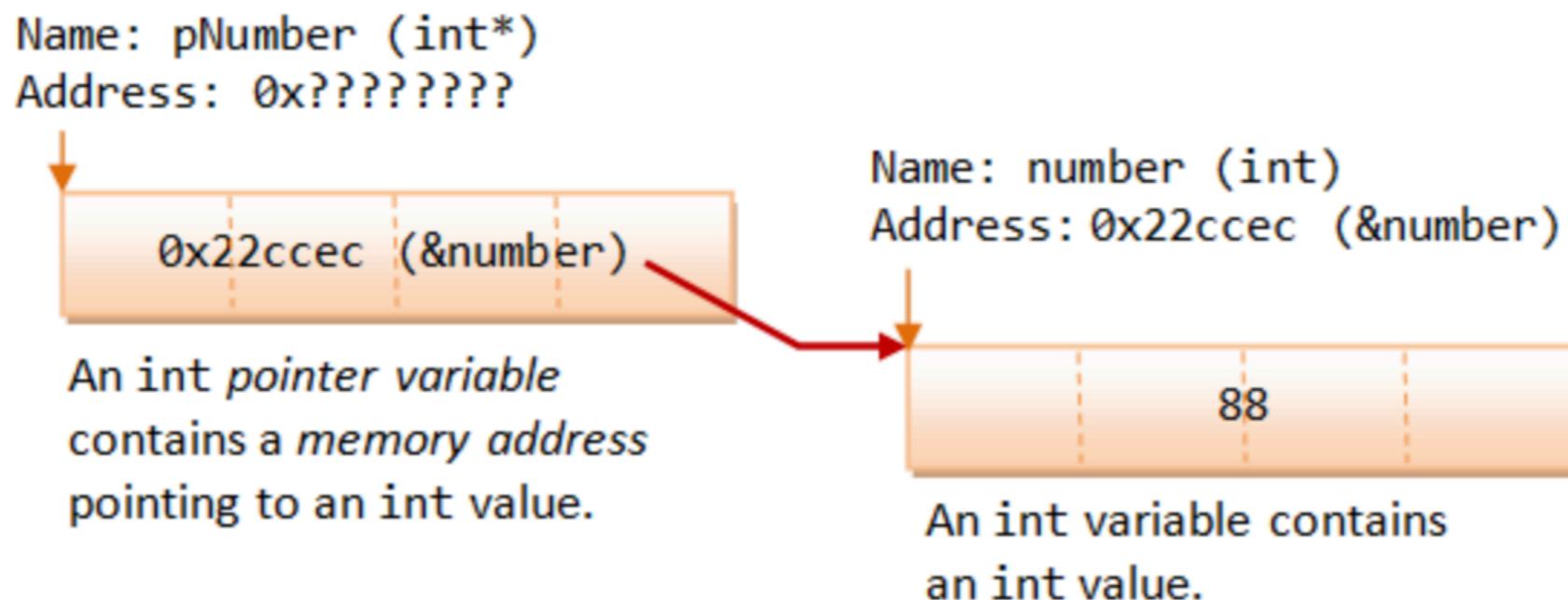


- Cuando un puntero se declara, no está inicializado. Cuidado! Una buena práctica es siempre declarar punteros inicializados, de lo contrario apuntan a una región en memoria indefinida
- Un puntero se puede inicializar dándole una referencia de memoria
- En C++, una referencia en memoria se declara mediante el operador &

```
int number = 88; // Un int con un valor  
int *pNumber; // Un puntero a un int sin inicializar  
pNumber = &number; // Ahora pNumber apunta a la dirección de number
```

```
int *pAnother = &number; // Otro puntero que apunta a number
```

Punteros y direcciones de memoria



Dereferenciación (anglicismo) de un puntero



- Un puntero se puede *dereferenciar*, es decir, **accesar el valor que existe en la región de memoria al cual el puntero apunta**
- Esto se realiza mediante el operador * actuando en el puntero

```
int number = 88; // Un int con un valor
int *pNumber = &number; // Un puntero a un int que apunta a la dirección de number

std::cout << pNumber << std::endl; // La dirección en memoria
std::cout << *pNumber << std::endl; // El valor al cual pNumber apunta: 88

*pNumber = 99; // Acceso el valor al cual pNumber apunta y le asigno otro valor

std::cout << *pNumber << std::endl; // El valor al cual pNumber apunta: 99
std::cout << number << std::endl; // El valor de number TAMBIEN CAMBIA: 99
```

Los punteros tienen tipos



- Los punteros también tienen tipos y **no pueden apuntar a variables de otro tipo**

```
int i = 88;
double d = 55.66;
int *i_ptr = &i;      // Puntero que apunta a la dirección de i
double *d_ptr = &d; // Puntero que apunta a la dirección de d

i_ptr = &d;    // ERROR, i_ptr no puede contener la dirección de otro tipo!
d_ptr = &i;    // ERROR
i_ptr = i;     // ERROR, el puntero contiene una dirección, no un valor!

int j = 99;
i_ptr = &j; // Ahora i_ptr apunta a la dirección de j
```



- Un puntero puede tener tipo NULL

```
int *i_ptr = 0; // Puntero sin inicialización
std::cout << *i_ptr << std::endl; // ERROR, STATUS_ACCESS_VIOLATION

int *p = NULL; // Puntero inicializado a ninguna dirección
```

Referencias



- El operador & designa una referencia en memoria
- El operador cambia su significado dependiendo de si se utiliza en una **expresión** o una **declaración**
- En una expresión, denota la referencia de memoria al objeto
- En una declaración (e.g., una función), denota una variable de referencia

```
int number = 88;           // Un int con nombre number
int &refNumber = number;   // Una referencia a la dirección de number
                          // refNumber y number se refieren al mismo valor

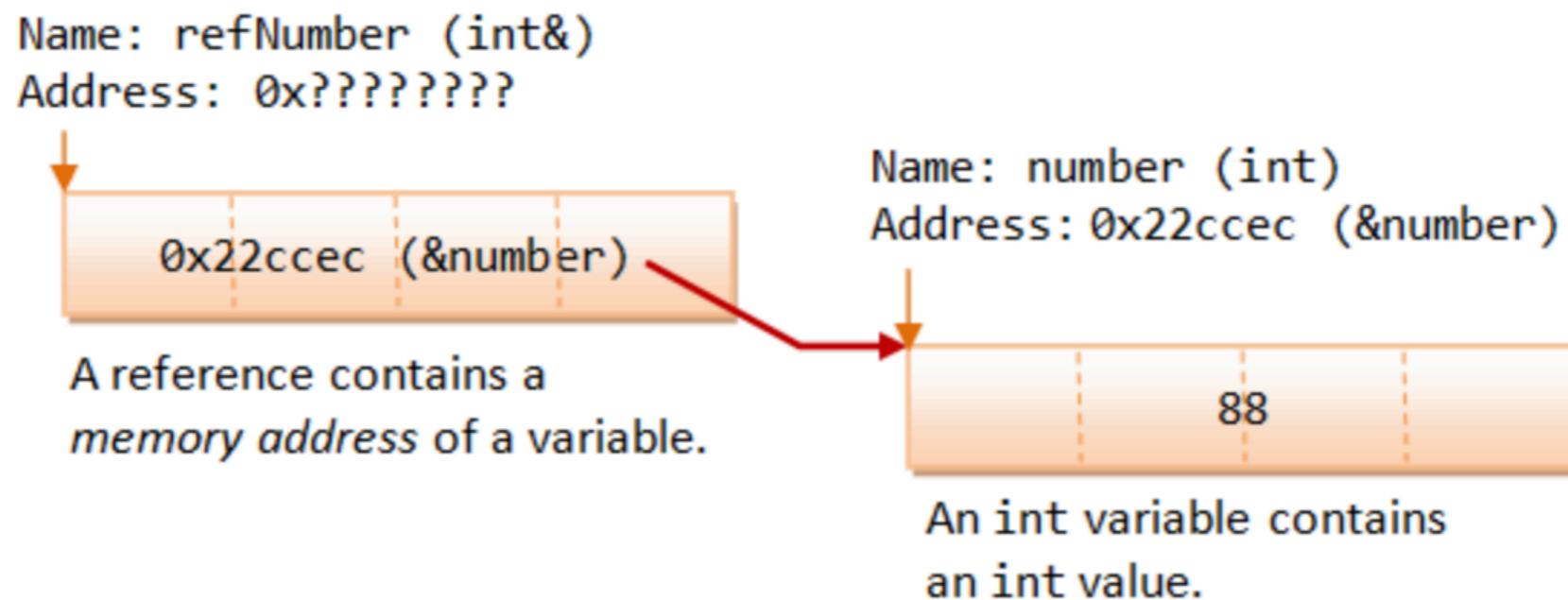
std::cout << number << std::endl;    // Imprime la variable: 88
std::cout << refNumber << std::endl; // Imprime el valor de la referencia: 88

refNumber = 99; // Cambia el valor de la variable a la cual la
                // referencia apunta

std::cout << refNumber << std::endl;
std::cout << number << std::endl; // El valor de la variable cambia: 99

number = 55; // Asigna un nuevo valor a number
std::cout << number << std::endl;
std::cout << refNumber << std::endl; // El valor de la referencia también cambia:
                                         // 55
```

Referencias



Punteros vs Referencias

- Los punteros y las referencias son equivalentes, con excepción de:



- Una referencia es **constante**. Cuando se establece una referencia a una variable u objeto, no se puede cambiar la referencia para referencia a otra variable u objeto
- Para obtener el valor al cual un puntero apunta, se necesita *dereferenciar* el puntero con el operador *. Por el otro lado, las referencias y dereferencias actuando sobre una referencia ocurre de manera implícita. Por ejemplo, refNumber es una referencia a un int, sin embargo, refNumber devuelve el valor de la variable. No se necesita dereferenciar la referencia

Punteros vs Referencias

```
1 #include <iostream>
2
3 int main(){
4
5     int number1 = 88, number2 = 22;
6
7     // Crear un puntero que apunta a number1
8     std::cout << "Punteros" << std::endl;
9     int *pNumber1 = &number1; // Referenciación explícita
10    *pNumber1 = 99;         // Dereferenciación explícita
11    std::cout << *pNumber1 << std::endl; // 99
12    std::cout << &number1 << std::endl; // Una dirección en memoria
13    std::cout << pNumber1 << std::endl; // La misma dirección en memoria
14    std::cout << &pNumber1 << std::endl; // La dirección en memoria DEL PUNTERO
15    pNumber1 = &number2; // El puntero se puede asignar a otra referencia
16
17    // Crear una referencia a number1
18    std::cout << "Referencias" << std::endl;
19    int &refNumber1 = number1; // Referencia implícita (NO SE PUEDE USAR &number1)
20    refNumber1 = 11;         // Dereferenciación implícita (NO SE USA *refNumber1)
21    std::cout << refNumber1 << std::endl; // 11
22    std::cout << &number1 << std::endl; // Una dirección en memoria
23    std::cout << &refNumber1 << std::endl; // La misma dirección en memoria
24    //refNumber1 = &number2; // Error! Las referencias no se pueden reasignar!
25                                // error: conversión inválida de 'int*' a 'int'
26    // Cuidado aquí!
27    refNumber1 = number2; // refNumber1 sigue siendo una referencia a number1!!!
28                                // Esto asigna el valor de number2 (22) a refNumber1 y number1!
29    number2++;
30    std::cout << refNumber1 << std::endl; // 22
31    std::cout << number1 << std::endl; // 22
32    std::cout << number2 << std::endl; // 23
33
34    return 0;
35 }
```

- `punteros_vs_referencias.cpp`

Argumentos de funciones (importante)

- └→ • Las funciones pueden recibir dos tipos de argumentos:
 - Variables y/o objetos
 - Referencias de variables y/o objetos
- └→ • Cuando una función recibe como argumento(s) una variable u objeto por su valor, dicho procedimiento se conoce como *pasar por valor*
 - Cuando una función recibe como argumento(s) una referencia de una variable u objeto, dicho procedimiento se conoce como *pasar por referencia*
- └→ • Las funciones también pueden devolver un valor o una referencia
- └→ • Dependiendo de cuál es el objetivo de una función, los argumentos se pasan por valor o por referencia
- └→ • Esto se puede hacer, en general, de tres formas diferentes en C++

Pasar por valor

```
1 #include <iostream>
2
3 int square(int n){
4
5     std::cout << "Dentro de la función square(): " << &n << std::endl; // Referencia de memoria
6     n *= n; // Esta expresión no tiene que ver con punteros. El operador * es de multiplicación
7         // Esta expresión es una forma compacta de escribir n = n * n
8         // Esto significa: multiplicar n por si mismo, y asignarlo a n
9         // En este scope, n es una copia del argumento!!!
10
11    return n; // El valor de n se retorna por valor.
12 }
13
14 int main(){
15
16     int number = 8;
17
18     std::cout << "Dentro de main(): " << &number << std::endl; // Referencia de memoria
19     std::cout << number << std::endl; // 8
20     std::cout << square(number) << std::endl; // 64
21     std::cout << number << std::endl; // 8 - number no cambia porque la función usa una copia
22                                     // y la retorna!
23
24 }
```

- pasar_por_valor.cpp

Pasar por referencia mediante puntero

```
1 #include <iostream>
2
3 // Nótese que esta función toma como argumento un puntero y no tiene valor de retorno!
4 // Comparar con pasar_por_valor.cpp
5 void square(int *pNumber){
6
7     std::cout << "Dentro de la función square(): " << pNumber << std::endl; // Una referencia de memoria
8     *pNumber *= *pNumber; // Mucho ojo con esta expresión
9         // Primero, dereferenciamos el puntero con el operador *
10        // Luego, multiplicamos el valor por si mismo
11        // Por último, asignamos el valor al cual la referencia apunta a este último valor:
12        // el resultado de la multiplicación
13 }
14
15 int main(){
16
17     int number = 8;
18     std::cout << "Dentro de main(): " << &number << std::endl; // Una referencia de memoria
19                                     // La misma que existe dentro de la función
20                                     // square()
21     std::cout << number << std::endl; // El valor de la referencia: 8
22     square(&number); // Referencia explícita de number
23                                     // Nótese que la función square espera una referencia de memoria,
24                                     // no un valor
25     std::cout << number << std::endl; // 64
26
27     return 0;
28 }
```

- `pasar_por_referencia_puntero.cpp`

Pasar por referencia mediante referencia

```
1 #include <iostream>
2
3 // Nótese que esta función toma como argumento una referencia y no tiene valor de retorno!
4 // Comparar con pasar_por_valor.cpp
5 void square(int &rNumber){
6
7     std::cout << "Dentro de la función square(): " << rNumber << std::endl; // Una referencia de memoria
8     rNumber *= rNumber; // Dereferenciación implícita
9         // Recordemos que accesar rNumber no accesa la referencia, sino su valor
10        // La diferencia con pasar_por_valor.cpp es que en esta función estamos
11            // usando la referencia del argumento, no su valor!
12 }
13
14 int main(){
15
16     int number = 8;
17     std::cout << "Dentro de main(): " << number << std::endl; // Una referencia de memoria
18                                     // La misma que existe dentro de la función
19                                     // square()
20     std::cout << number << std::endl; // El valor de la referencia: 8
21     square(number); // Referencia implícita de number
22         // La función toma la referencia del argumento, no su valor!
23         // Aquí hay que tener cuidado. Las referencias se inicializan durante
24             // la declaración.
25         // En el caso de la función square(), la referencia se inicia cuando la función
26             // es invocada, con la referencia del argumento de la función
27             // Esto puede ser confuso. square(&number) resulta en un error!
28             // Esto se debe a que la referenciación y dereferenciación
29                 // ocurren de manera implícita
30     std::cout << number << std::endl; // 64
31
32     return 0;
33 }
```

- pasar_por_referencia_referencia.cpp

Punteros vs Referencias: argumentos de funciones

- Existen diversas formas de pasar argumentos a funciones



- Mediante la denotación `const`, un argumento se puede hacer explícitamente inmutable
- En general, si se desea modificar un valor o una referencia depende intrínsecamente de cual es el objetivo deseado de la función
- Es muy común no desear que una función realice copias internas
- Copias internas pueden consumir muchos recursos de memoria

Punteros vs Referencias

- En conclusión



- Una referencia es algo similar a un puntero, con la diferencia de que la referencia devuelve el valor al cual la referencia apunta y no la referencia!
- Una referencia es básicamente un **alias** de algún objeto

Punteros vs Referencias

- En conclusión
 - Usar referencias **si es posible**
 - Usar punteros **solamente si es absolutamente necesario**

Memoria

- C++ permite el manejo de memoria desde bajo nivel
 - **El manejo correcto de los recursos memoria es uno de los conceptos más importantes en este curso**

Memoria: Segmentos globales y de código

```
1 #include <iostream>
2
3 // Global Segment: Global variables are stored here
4 int globalVar = 42;
5
6 // Code Segment: Functions and methods are stored here
7 int add(int a, int b) {
8     return a + b;
9 }
10
11 int main() {
12     // Code Segment: Calling the add function
13     int sum = add(globalVar, 10);
14
15     std::cout << "Sum: " << sum << std::endl;
16     return 0;
17 }
```

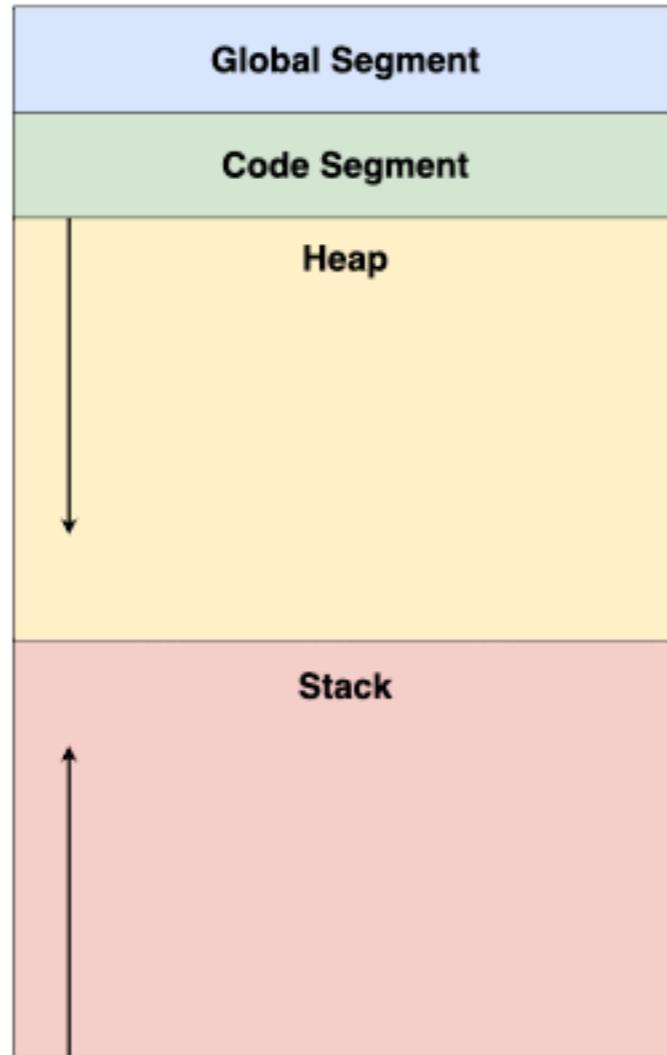
- **Una buena práctica es NUNCA usar variables globales**

Global Segment	Variable and its value: globalVar = 42
Code Segment	Complete function definition: add(a, b)

Memoria: Stack y heap (importante)

**La diferencia entre
memoria estática y
memoria dinámica**

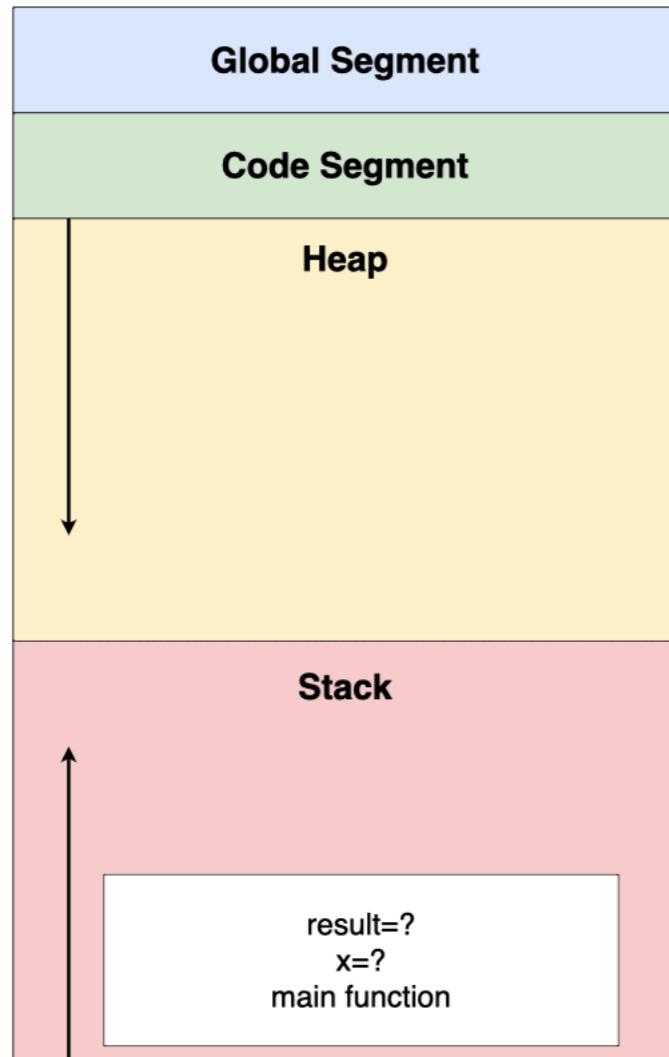
Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19     return 0;
20 }
```

The stack segment is empty

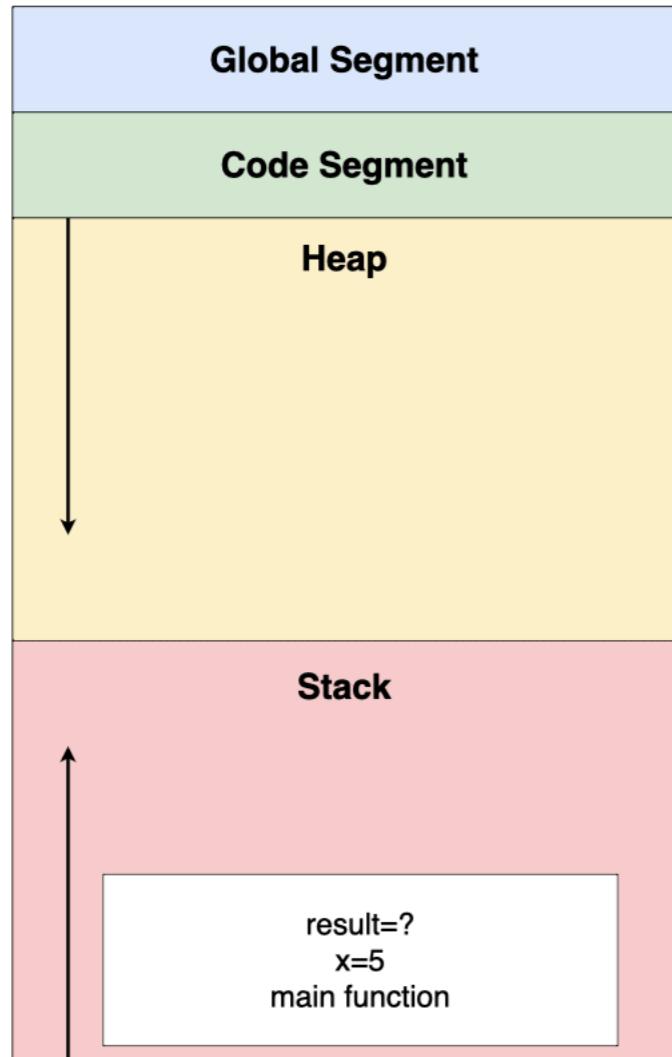
Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19     return 0;
20 }
```

A new stack frame is created for the main function

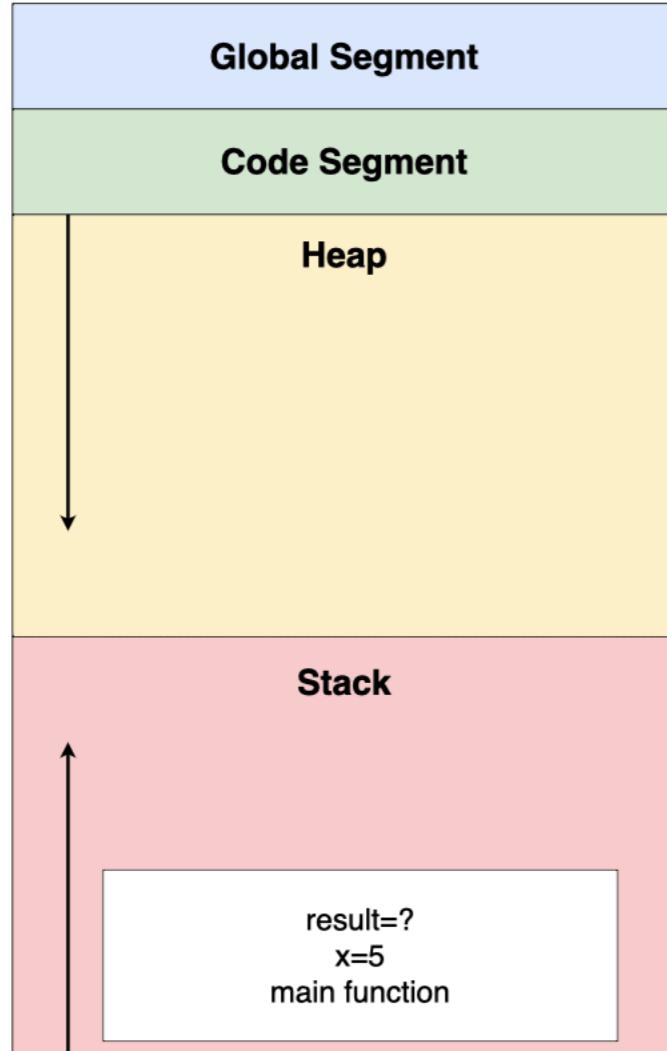
Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19     return 0;
20 }
```

In the stack frame for the main function, the local variable x now has a value of 5

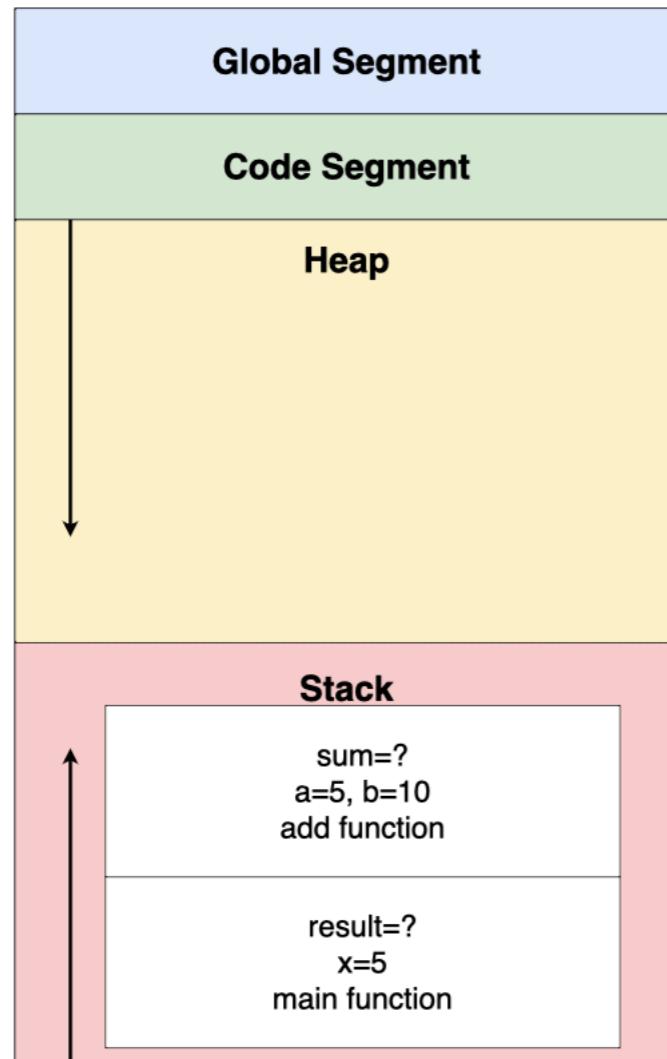
Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19     return 0;
20 }
```

A function call to add with actual parameters as (5, 10) is made

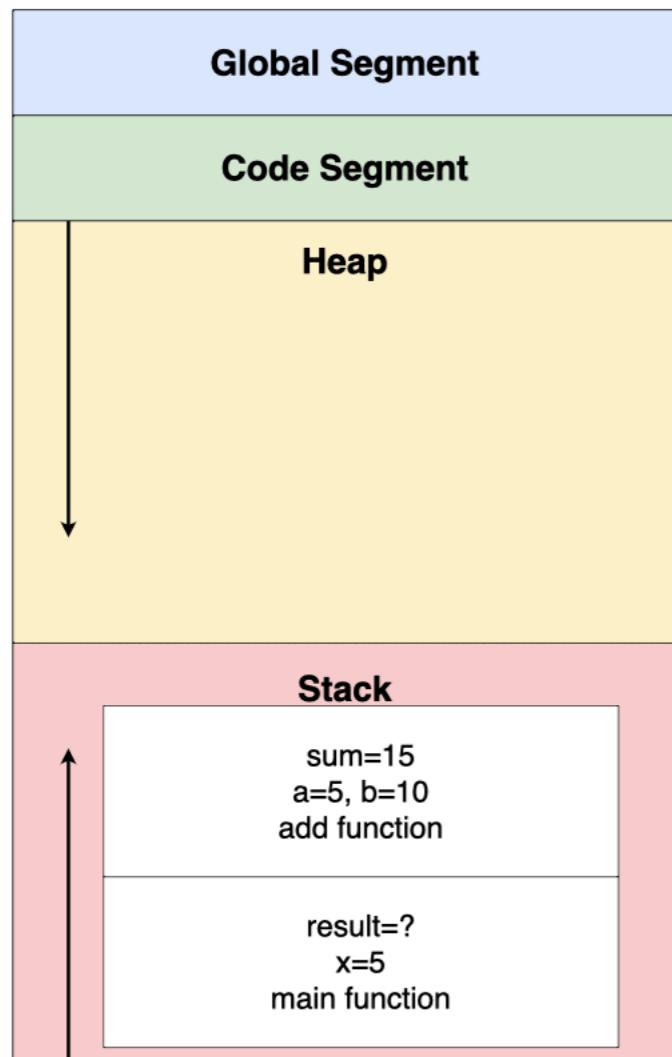
Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19 }
20 }
```

The control is transferred to the add function, a new stack frame for add function with local variables a, b, and sum is created

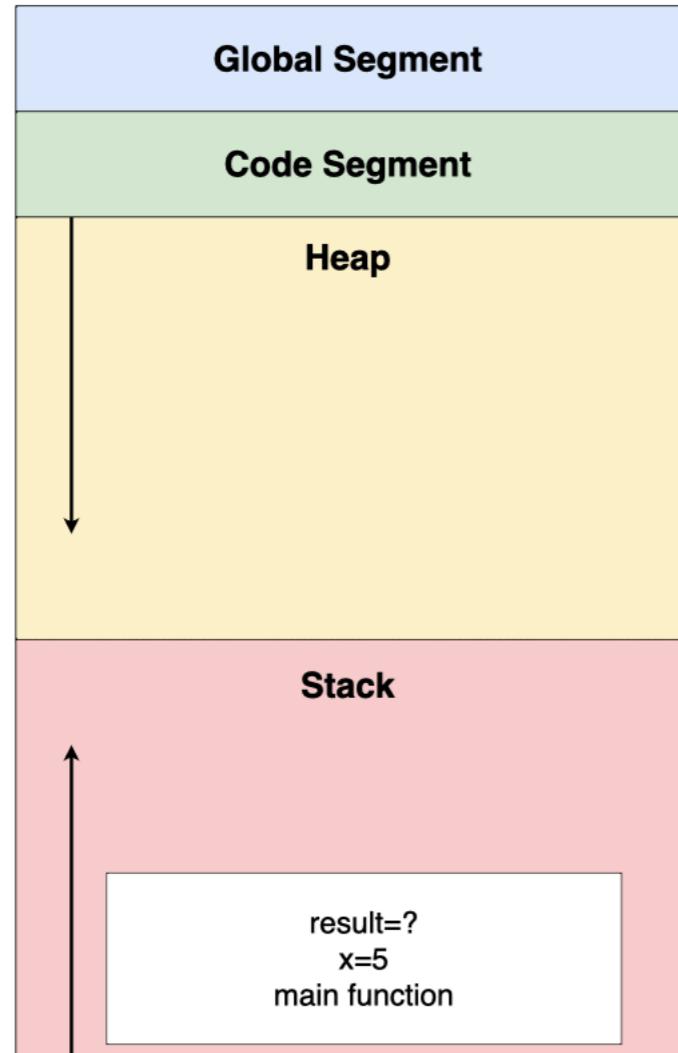
Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19     return 0;
20 }
```

The sum variable on stack frame for the add function is assigned the result of $a + b$

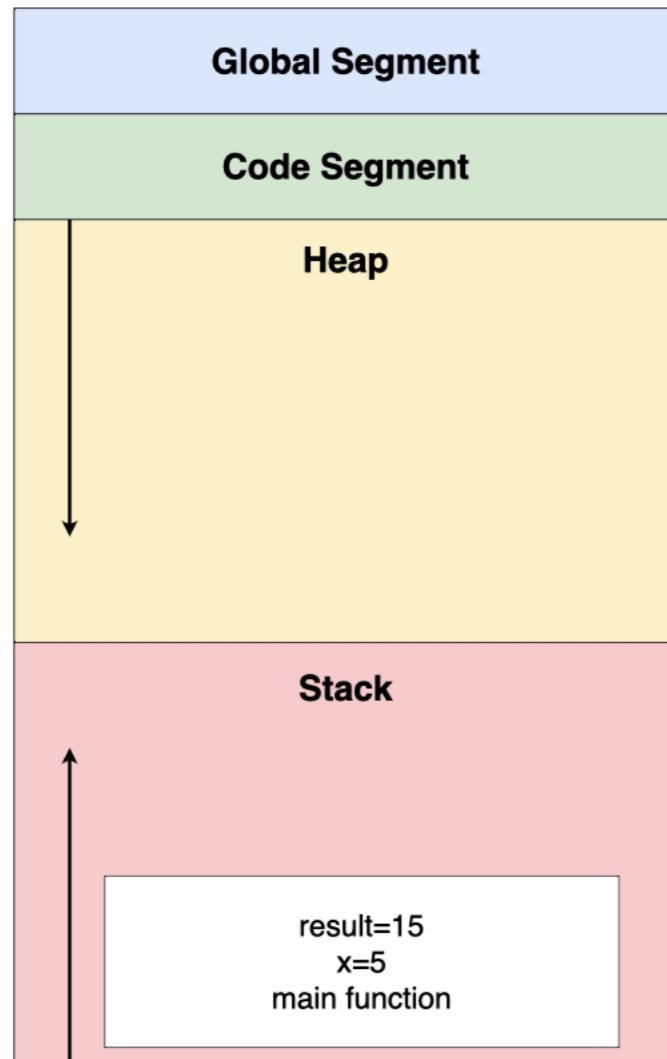
Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19     return 0;
20 }
```

The add function completes its task and its stack frame gets destroyed

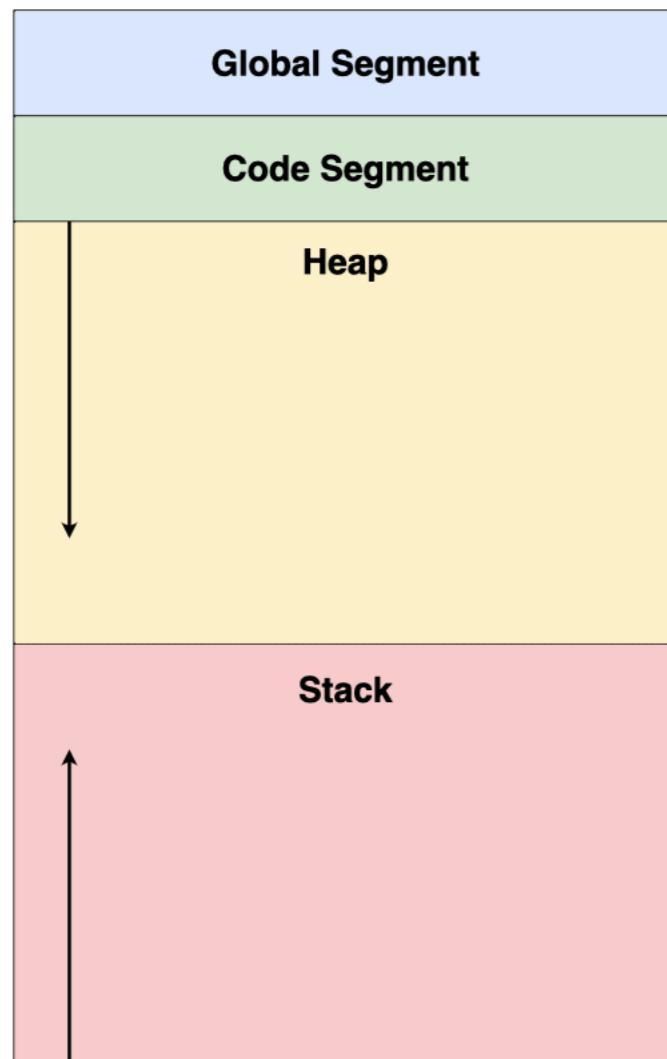
Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19     return 0;
20 }
```

The stack frame for the main function with variable result stores the value returned from the add function

Memoria estática: Stack (importante)



```
1 #include <iostream>
2
3 // A simple function to add two numbers
4 int add(int a, int b) {
5     // Local variables (stored in the stack)
6     int sum = a + b;
7     return sum;
8 }
9
10 int main() {
11     // Local variable (stored in the stack)
12     int x = 5;
13
14     // Function call (stored in the stack)
15     int result = add(x, 10);
16
17     std::cout << "Result: " << result << std::endl;
18
19     return 0;
20 }
```

The main function block also gets destroyed, after displaying the value of result (not shown here), and the stack segment is empty again

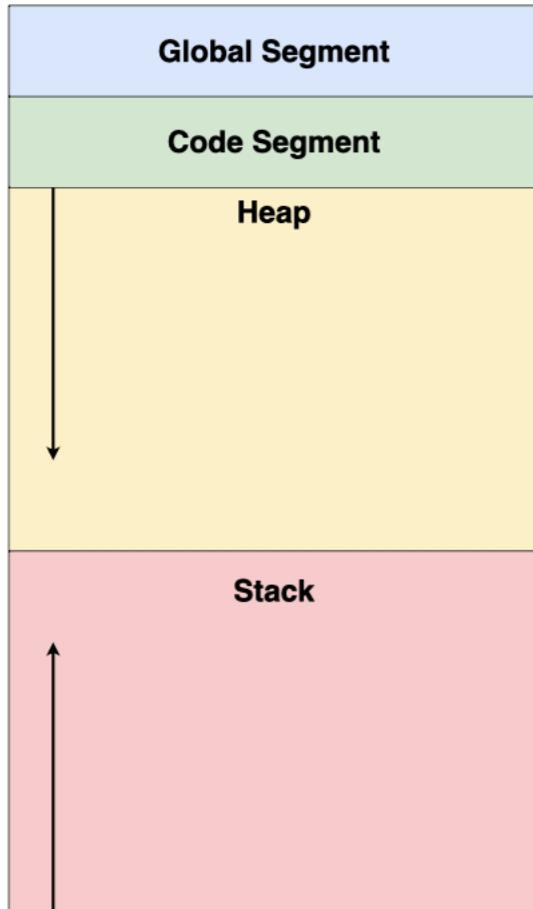
Características del stack

- Tamaño fijo
 - └→ • La memoria stack tiene tamaño fijo. Su tamaño se determina al inicio de la ejecución de un programa y su tamaño no cambia (compilador)
- Velocidad
 - └→ • Los segmentos de memoria del stack son continuos y su asignación y designación es usualmente rápida mediante referencias manejadas por el kernel
- Almacenamiento para información de control y variables
 - └→ • La memoria de stack se utiliza para almacenar, e.g., información de control, variables locales, argumentos de funciones y direcciones de retorno
- Acceso limitado
 - └→ • El stack solo se puede accesar mediante llamados a funciones activas
- Manejo automático
 - └→ • El manejo del stack no requiere esfuerzo del programador, sino del sistema operativo

Importante

El stack tiene tamaño fijo en el momento de la compilación, de manera tal que no se puede utilizar para asignar segmentos de memoria **cuyo tamaño se desconoce al momento de la compilación**. De ahí su nombre memoria estática. Para asignación de memoria dinámica, se usa el **heap**

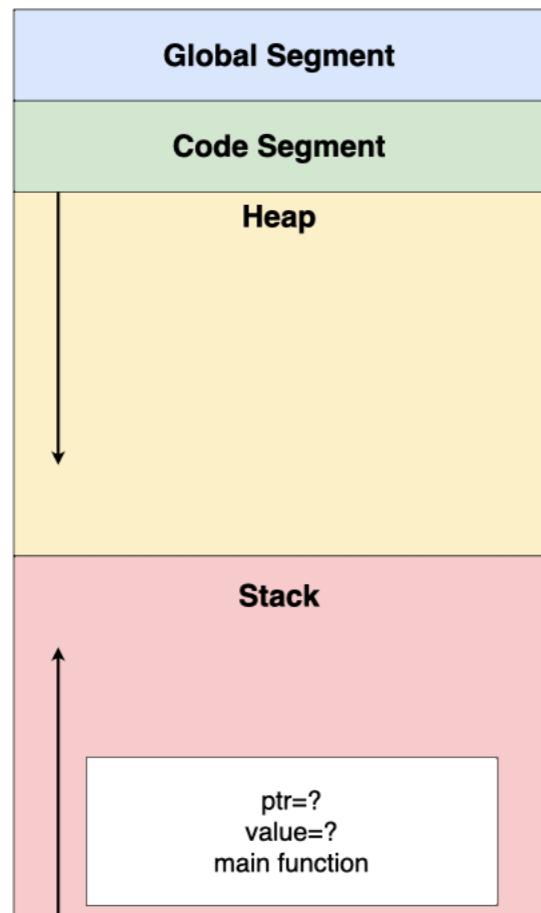
Memoria dinámica: Heap (importante)



```
1 #include <iostream>
2
3 int main() {
4     // Stack: Local variable 'value' is stored on the stack
5     int value = 42;
6
7     // Heap: Allocate memory for a single integer on the heap
8     int* ptr = new int;
9
10    // Assign the value to the allocated memory and print it
11    *ptr = value;
12    std::cout << "Value: " << *ptr << std::endl;
13
14    // Deallocate memory on the heap
15    delete ptr;
16
17    return 0;
18 }
```

The stack and heap segments are empty

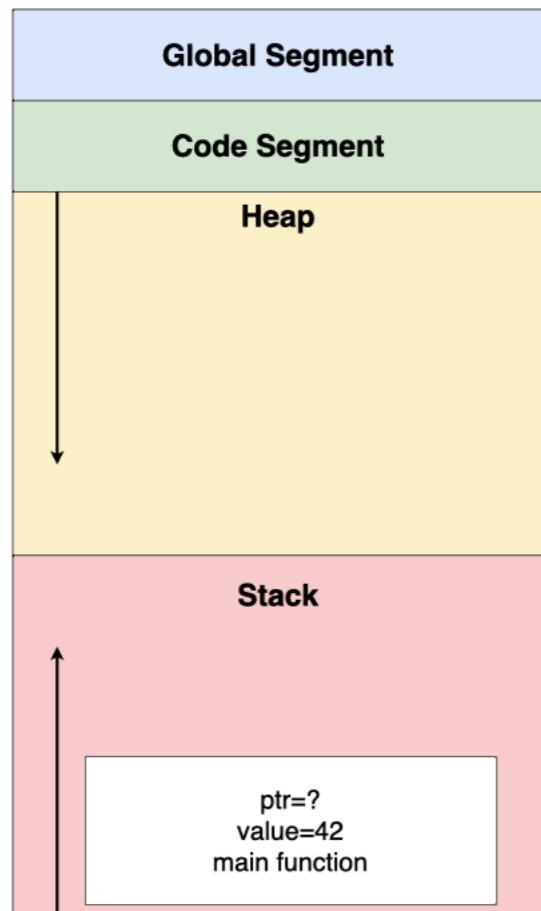
Memoria dinámica: Heap (importante)



```
1 #include <iostream>
2
3 int main() {
4     // Stack: Local variable 'value' is stored on the stack
5     int value = 42;
6
7     // Heap: Allocate memory for a single integer on the heap
8     int* ptr = new int;
9
10    // Assign the value to the allocated memory and print it
11    *ptr = value;
12    std::cout << "Value: " << *ptr << std::endl;
13
14    // Deallocate memory on the heap
15    delete ptr;
16
17    return 0;
18 }
```

A new stack frame is created for the main function

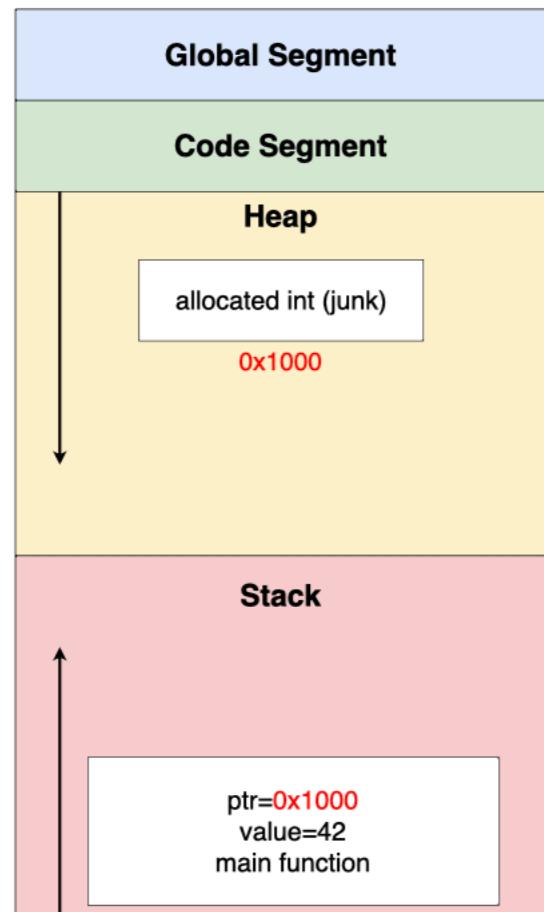
Memoria dinámica: Heap (importante)



```
1 #include <iostream>
2
3 int main() {
4     // Stack: Local variable 'value' is stored on the stack
5     int value = 42;
6
7     // Heap: Allocate memory for a single integer on the heap
8     int* ptr = new int;
9
10    // Assign the value to the allocated memory and print it
11    *ptr = value;
12    std::cout << "Value: " << *ptr << std::endl;
13
14    // Deallocate memory on the heap
15    delete ptr;
16
17    return 0;
18 }
```

A local variable `value` is assigned the value 42

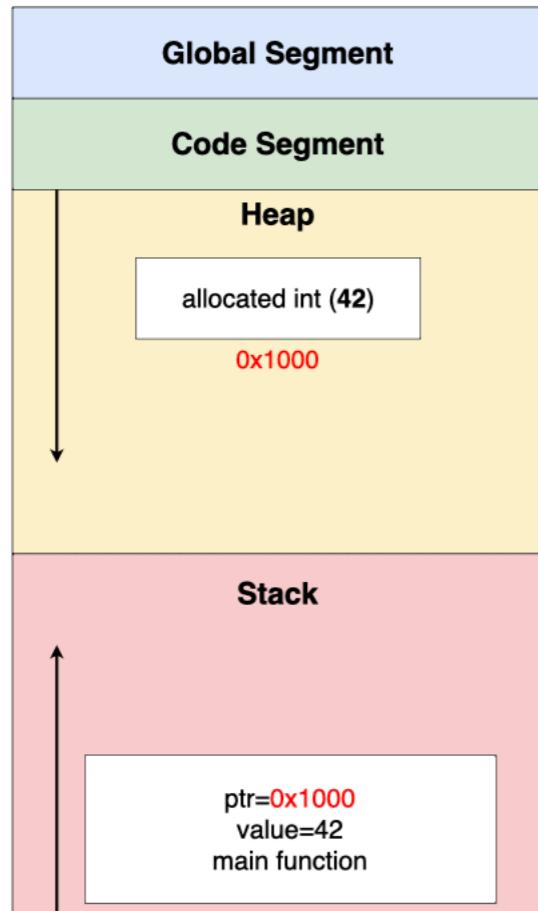
Memoria dinámica: Heap (importante)



```
1 #include <iostream>
2
3 int main() {
4     // Stack: Local variable 'value' is stored on the stack
5     int value = 42;
6
7     // Heap: Allocate memory for a single integer on the heap
8     int* ptr = new int;
9
10    // Assign the value to the allocated memory and print it
11    *ptr = value;
12    std::cout << "Value: " << *ptr << std::endl;
13
14    // Deallocate memory on the heap
15    delete ptr;
16
17    return 0;
18 }
```

A pointer variable `ptr` is allocated memory on the heap, and the address of the allocated heap memory (i.e., `0x1000`) is stored in the pointer `ptr`

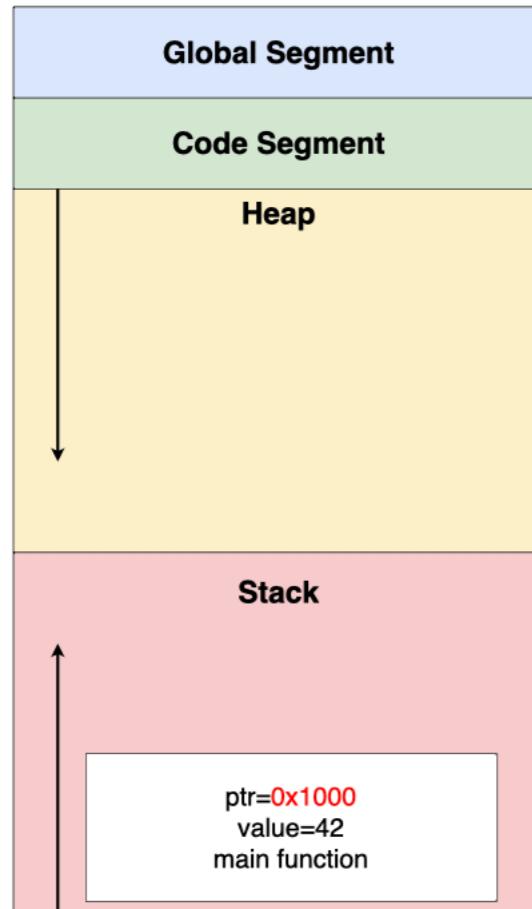
Memoria dinámica: Heap (importante)



```
1 #include <iostream>
2
3 int main() {
4     // Stack: Local variable 'value' is stored on the stack
5     int value = 42;
6
7     // Heap: Allocate memory for a single integer on the heap
8     int* ptr = new int;
9
10    // Assign the value to the allocated memory and print it
11    *ptr = value;
12    std::cout << "Value: " << *ptr << std::endl;
13
14    // Deallocate memory on the heap
15    delete ptr;
16
17    return 0;
18 }
```

The value stored in the value variable (i.e., 42) is assigned to the memory location pointed to by ptr (heap address 0x1000)

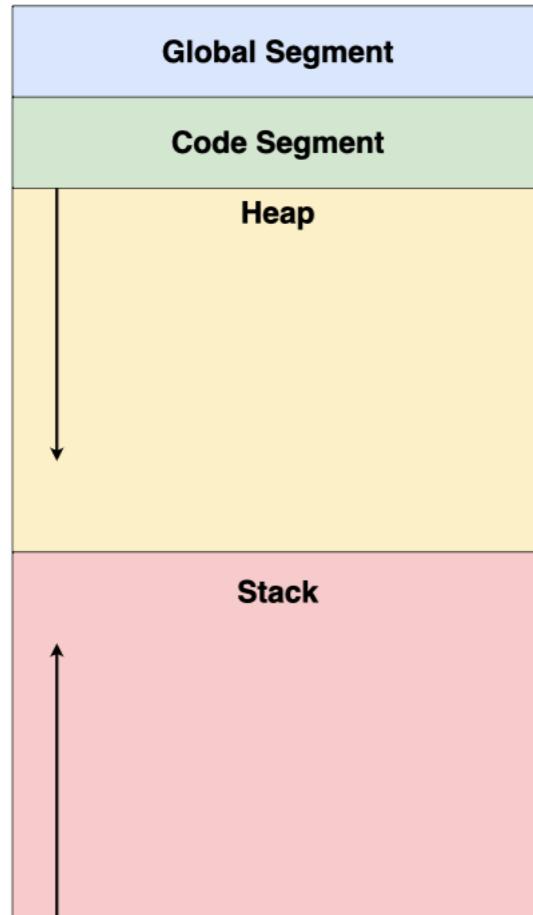
Memoria dinámica: Heap (importante)



```
1 #include <iostream>
2
3 int main() {
4     // Stack: Local variable 'value' is stored on the stack
5     int value = 42;
6
7     // Heap: Allocate memory for a single integer on the heap
8     int* ptr = new int;
9
10    // Assign the value to the allocated memory and print it
11    *ptr = value;
12    std::cout << "Value: " << *ptr << std::endl;
13
14    // Deallocate memory on the heap
15    delete ptr;
16
17    return 0;
18 }
```

The memory allocated on the heap at address 0x1000 is deallocated

Memoria dinámica: Heap (importante)



```
1 #include <iostream>
2
3 int main() {
4     // Stack: Local variable 'value' is stored on the stack
5     int value = 42;
6
7     // Heap: Allocate memory for a single integer on the heap
8     int* ptr = new int;
9
10    // Assign the value to the allocated memory and print it
11    *ptr = value;
12    std::cout << "Value: " << *ptr << std::endl;
13
14    // Deallocate memory on the heap
15    delete ptr;
16
17    return 0;
18 }
```

The main function's stack frame is popped from the stack (after displaying the value of result), and the stack and heap segments are empty again

Características del heap

- Tamaño indeterminado
 - └→ • El tamaño del heap asignado cambia durante la ejecución de un programa
- Velocidad menor al stack
 - └→ • Asignar y desasignar memoria del heap es más lento debido a que incurre encontrar marcos de memoria y fragmentación
- Almacenamiento para objetos dinámicos
 - └→ • El heap se utiliza para estructuras de datos y objetos con tiempos de vida dinámicos
- Data persistente
 - └→ • Lo que se almacena en el heap se mantiene en el heap hasta que se desasigna manualmente (*memory leaks*)
- Manejo manual
 - └→ • Requiere destreza del programador

Cuando utilizar el stack o el heap?

- De manera sencilla: usar siempre el stack excepto cuando
 - • Existe la necesidad de almacenar objetos, estructuras de datos o arreglos dinámicos cuyo tiempo de vida no puede ser determinado durante la compilación o durante un llamado a una función
 - Los requerimientos de memoria son muy grandes o cuando debemos compartir datos entre diferentes secciones del programa
 - No conocemos el tamaño de un arreglo en el momento de la compilación
 - La asignación de memoria persiste más allá del scope de una sola función