



Funciones

Universidad Católica Boliviana

MSc, José Jesús Cabrera Pantoja

Outline

- Repaso Vectores
- Integrated Development Enviroment (IDE)
- Funciones
- Argumentos
- Valores de retorno
- Scope

Vectores Recap

For loops en Vectores

- Al igual que en las strings podemos pasarnos elemento por elemento de los vectores utilizando un loop



```
1 for (int month_index = 0; month_index < month_lengths.size(); ++month_index) {  
2     cout << "There are "s << month_lengths[month_index]  
3         << " days "s  
4         << " in month "s << (month_index + 1) << endl;  
5 }
```

For loops en Vectores

- Al igual que en las strings podemos pasarnos elemento por elemento de los vectores utilizando un loop
- Este ciclo se ve feo, se puede cometer errores, se repite varias veces la variable month_index, etc.



```
1 for (int month_index = 0; month_index < month_lengths.size(); ++month_index) {  
2     cout << "There are "s << month_lengths[month_index]  
3         << " days "s  
4         << " in month "s << (month_index + 1) << endl;  
5 }
```

For loops en Vectores

- Al igual que en las strings podemos pasarnos elemento por elemento de los vectores utilizando un loop
- Este ciclo se ve feo, se puede cometer errores, se repite varias veces la variable month_index, etc.



```
1 for (int month_index = 0; month_index < month_lengths.size(); ++month_index) {  
2     cout << "There are "s << month_lengths[month_index]  
3         << " days "s  
4         << " in month "s << (month_index + 1) << endl;  
5 }
```

- Buenas noticias! Hay una mejor forma de hacerlo 😊. Se llama **range-based for**

Range-based for

- El ejemplo utiliza un **range-based for** para recorrer el vector. La variable de **length** se ejecutará a través de todos sus elementos en secuencia:



```
1 cout << "Month lengths are:"s;  
2  
3 // Los valores de la variable length por orden  
4 // tendra los elementos del vector month_length  
5 for (int length : month_lengths) {  
6     cout << " "s << length;  
7 }  
8 cout << endl;
```

Range-based for

- El ejemplo utiliza un **range-based for** para recorrer el vector. La variable de **length** se ejecutará a través de todos sus elementos en secuencia:

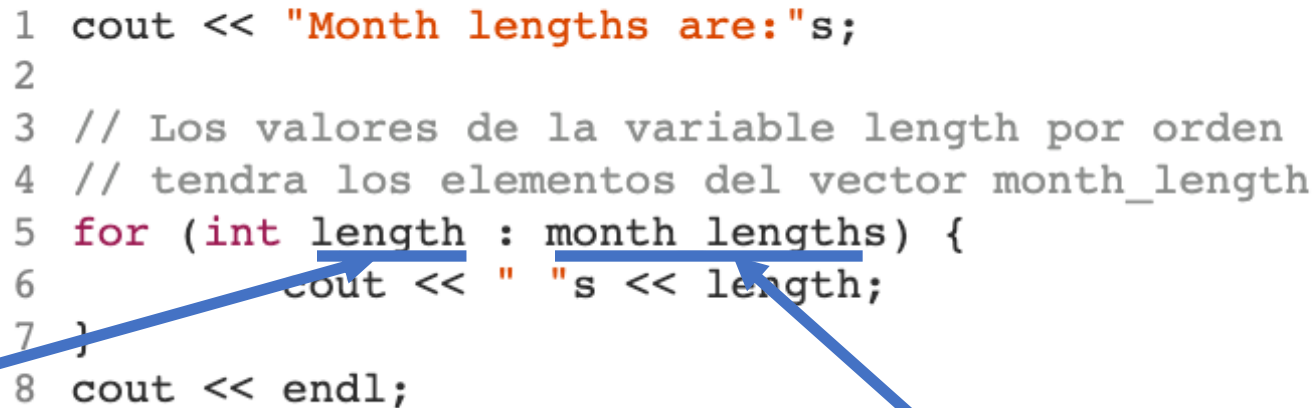


```
1 cout << "Month lengths are:"s;  
2  
3 // Los valores de la variable length por orden  
4 // tendra los elementos del vector month_length  
5 for (int length : month_lengths) {  
6     cout << " "s << length;  
7 }  
8 cout << endl;
```

Variable
length que
guardara los
elementos
del vector

Range-based for

- El ejemplo utiliza un **range-based for** para recorrer el vector. La variable de **length** se ejecutará a través de todos sus elementos en secuencia:



```
1 cout << "Month lengths are:"s;
2
3 // Los valores de la variable length por orden
4 // tendra los elementos del vector month_length
5 for (int length : month_lengths) {
6     cout << " "s << length;
7 }
8 cout << endl;
```

Variable
length que
guardara los
elementos
del vector

Vector
utilizado

Range-based for

- El ejemplo utiliza un **range-based for** para recorrer el vector. La variable de **length** se ejecutará a través de todos sus elementos en secuencia:



```
1 cout << "Month lengths are:"s;  
2  
3 // Los valores de la variable length por orden  
4 // tendra los elementos del vector month_length  
5 for (int length : month_lengths) {  
6     cout << " "s << length;  
7 }  
8 cout << endl;
```

- Salida:




```
1 Month lengths are: 31 28 31 30 31 30 31 31 30 31 30 31
```

Keyword **auto**

- Hasta ahora se ha visto cómo almacenar tipos básicos y vectores que contienen esos tipos.
- Mientras practicaba la declaración de variables, en cada caso indicó el tipo de variable.
- Sin embargo, es posible que C++ haga una inferencia de tipo automática, usando la palabra clave **auto**.

Keyword **auto**

- Sin embargo, es posible que C++ haga una inferencia de tipo automática, usando la palabra clave **auto**.



```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main() {
7     auto i = 5;
8     auto v_6 = {1, 2, 3};
9     cout << "Variables declared and initialized without explicitly stating type!" << "\n";
10 }
```

Keyword **auto**

- Sin embargo, es posible que C++ haga una inferencia de tipo automática, usando la palabra clave **auto**.

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main() {
7     auto i = 5;
8     auto v = {1, 2, 3};
9     cout << "Variables declared and initialized without explicitly stating type!" << "\n";
10 }
```

Le decimos al compilador que adivine que tipo de dato es esta variable (en este caso entero, int)

Keyword **auto**

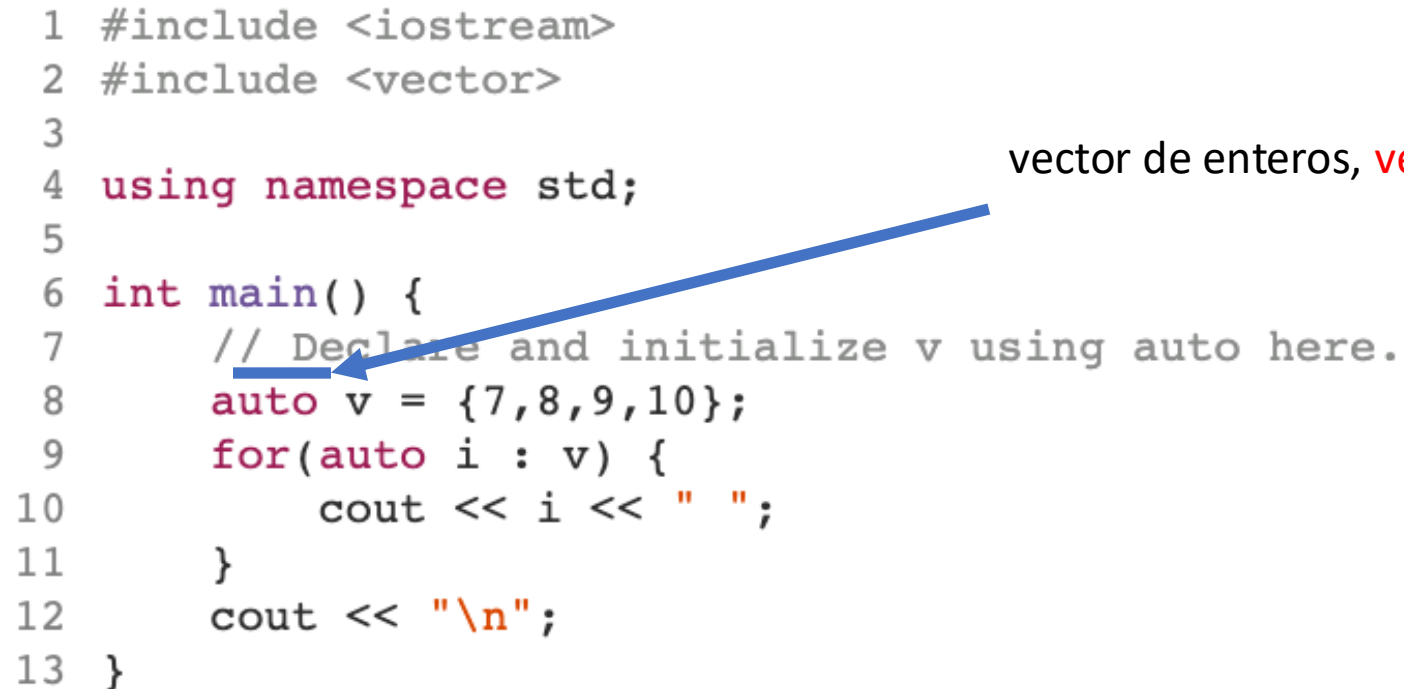
- Sin embargo, es posible que C++ haga una inferencia de tipo automática, usando la palabra clave **auto**.

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main() {
7     auto i = 5;
8     auto v_6 = {1, 2, 3};
9     cout << "Variables declared and initialized without explicitly stating type!" << "\n";
10 }
```

Le decimos al compilador que adivine que tipo de dato es esta variable (en este caso vector de enteros, `vector<int>`)

Keyword **auto**

- Incluso se puede utilizar **auto** en el **range-based for** ciclo



```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main() {
7     // Declare and initialize v using auto here.
8     auto v = {7,8,9,10};
9     for(auto i : v) {
10         cout << i << " ";
11     }
12     cout << "\n";
13 }
```

vector de enteros, **vector<int>**

Keyword **auto**

- Incluso se puede utilizar **auto** en el **range-based for** ciclo

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 int main() {
7     // Declare and initialize v using auto here.
8     auto v = {7,8,9,10};
9     for(auto i : v) {
10         cout << i << " ";
11     }
12     cout << "\n";
13 }
```

vector de enteros, **vector<int>**

Que tipo de dato tendrá la variable **i**?

Keyword **auto**

- Es útil declarar manualmente el tipo de una variable si desea que el tipo de variable sea claro para el lector de su código, o si desea ser explícito sobre la precisión numérica que se utiliza.
- C++ tiene varios tipos de números con diferentes niveles de precisión, y es posible que esta precisión no quede clara a partir del valor que se asigna.
- No es bueno abusar del uso de auto, se debe utilizar lo menos posible o únicamente cuando la precisión no es necesaria.

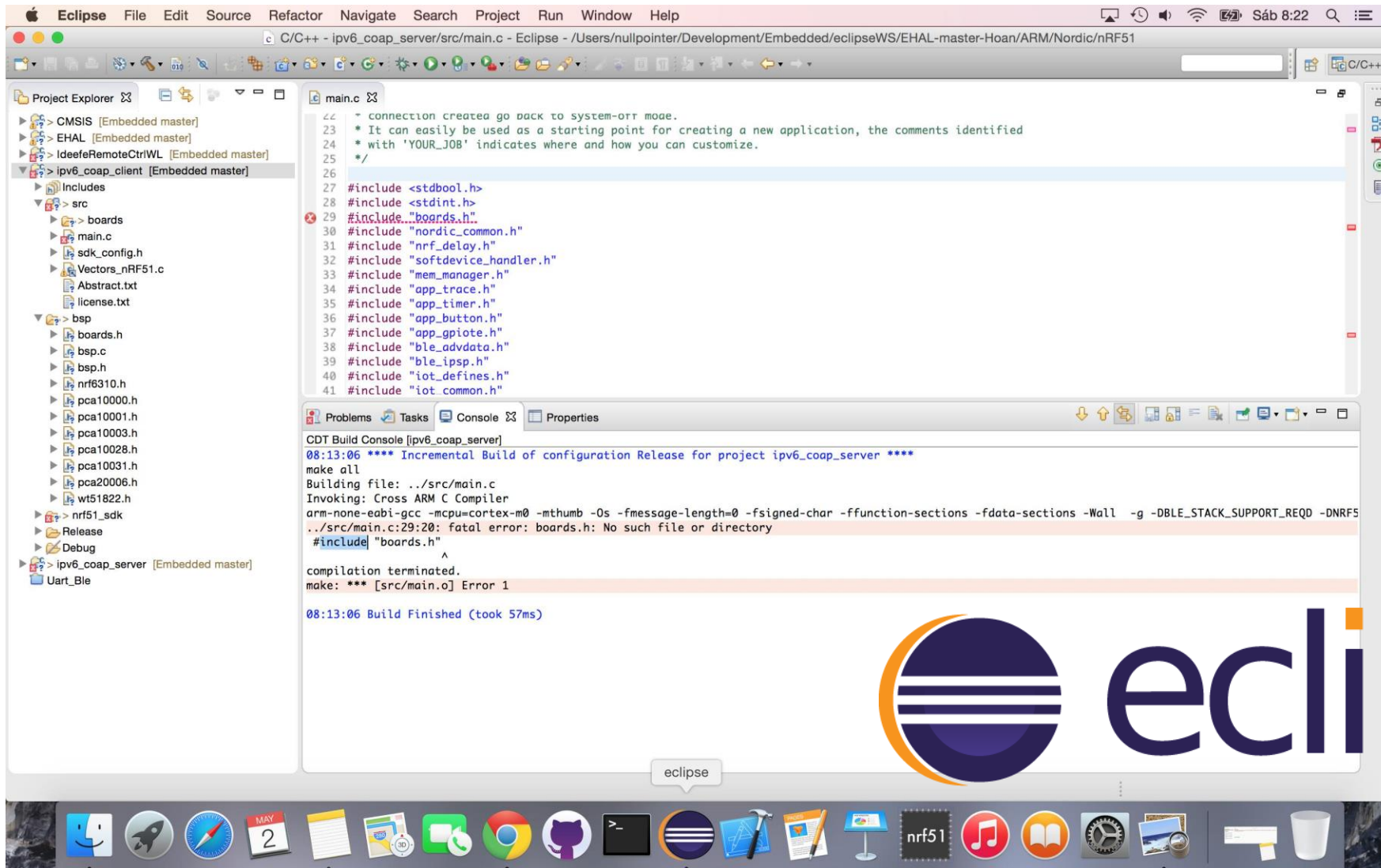
Entorno de desarrollo integrado (IDE)

- Ayuda a los programadores de computadoras a escribir, probar y depurar su código.
- Por lo general, incluye un editor de código con características como resaltado de sintaxis, autocompletado y formato de código.
- También tiene herramientas para compilar o interpretar código y ejecutarlo dentro del IDE.
- Pesado
- Hay que comprender sus funcionalidades

Entorno de desarrollo integrado (IDE)

- Existen variedades de IDE estas dependen del lenguaje de programación que se esta utilizando. Por ejemplo, para C++, C, C#:
 - Eclipse IDE – C/C++, C#, Java – inicial
 - CLion IDE (Jetbrains) – C/C++ - mas avanzado
 - Visual Studio – C/C++, C# - intermedio
- Para otros lenguajes de programación hay que identificar el IDE mas utilizado.

IDE - Eclipse

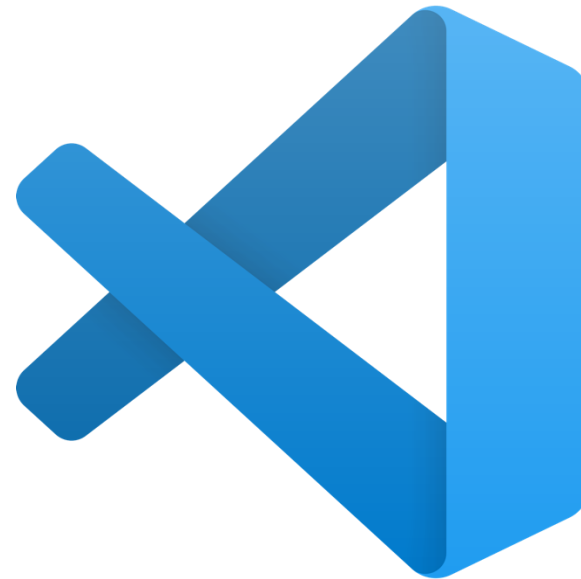


Editores de texto

- Un editor de código tiene características como resaltado de sintaxis, autocompletado y formato de código.
- Para compilar el código o interpretarlo es necesario realizar algunas configuraciones adicionales.
- Por ejemplo, para C++ es necesario configurar el compilador! Esto es posible con algunas herramientas adicionales, como ser CMake.
- El editor de texto mas usado actualmente es **Vscode!!**
 - Ligero
 - Flexible

Vscode

- Sin embargo, **vscode** puede comportarse como un IDE.
- Esto se debe a que tiene extensiones para configurarlo dependiendo del lenguaje de programación que se este usando.
- Lo configuraremos para usarlo con C++.



Funciones

- Agrupar el código según un criterio
- Separar el código por propósito
- Reutilizar el código en varias partes del programa

Separar el codigo

Verificar el password



Manejar errores



Calcular el total a pagar



Mostrar el menú de un
restaurante



Creando funciones

Nos ayuda a

- Organizar
- Modificar
- Entender
- NO reinventar la rueda
- **DONT REPEAT YOUR SELF**

Nombre de nuestra funcion



Creando funciones

Otros nombres

- Modulos
- Metodos

Nombre de nuestra funcion



Creando funciones



```
1 # Mas codigo
2 ...
3 instruccion 1
4 instruccion 2
5 instruccion 3
6 instruccion 4
7 instruccion 5
8 instruccion 6
9 ...
10 # Mas codigo
```

Prototipo de la funcion

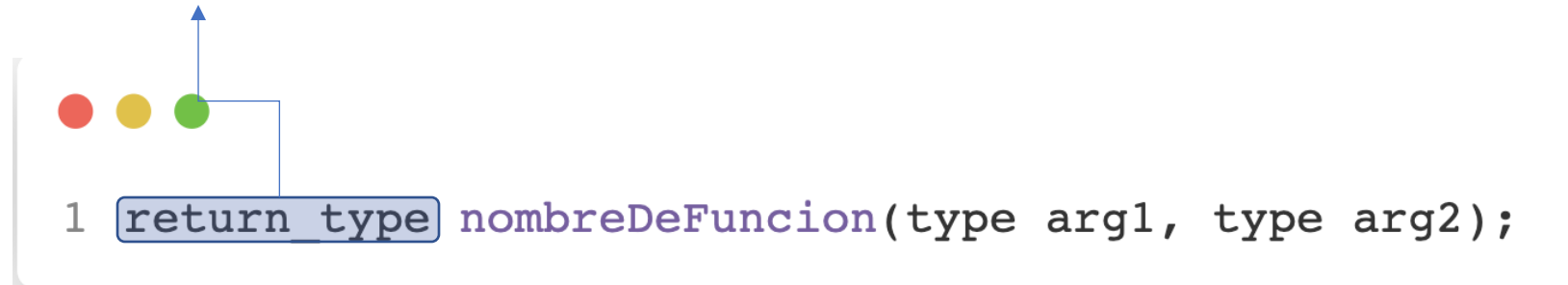
- Un prototipo de función es una declaración de una función que especifica su nombre, tipo de retorno y tipos de parámetros, pero no su cuerpo.
- Proporciona al compilador información sobre la firma de la función para que pueda llamarse correctamente.



```
1 return_type nombreDeFuncion(type arg1, type arg2);
```

Prototipo de la funcion

Lo que retorna la función,
esto puede ser un int,
double, string (cualquier tipo
de dato)

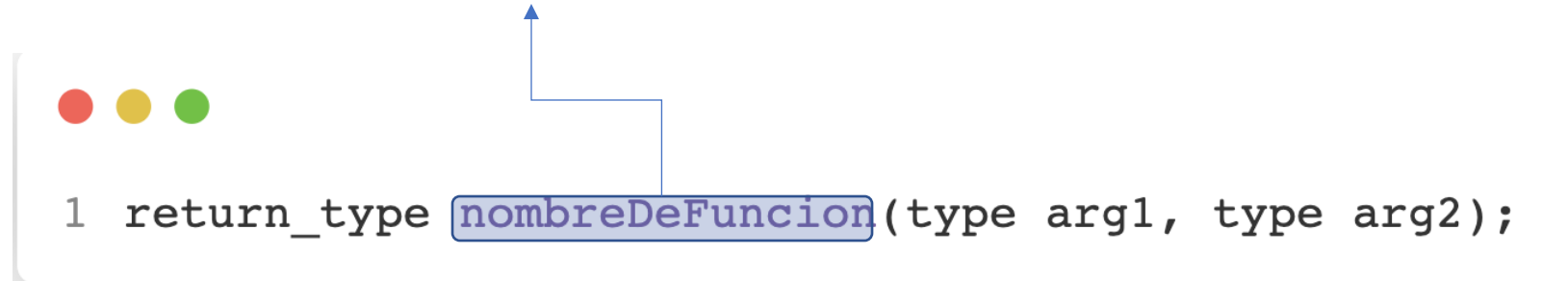


1 `return_type` nombreDeFuncion(type arg1, type arg2);

The diagram illustrates the components of a function prototype. A blue box highlights the `return_type` in the code snippet. A blue arrow originates from the green dot of a three-dot menu (red, yellow, green) and points to the text 'Lo que retorna la función, esto puede ser un int, double, string (cualquier tipo de dato)', which explains the possible data types for the return value.

Prototipo de la funcion

Nombre de la función. Esto es su firma, indentificador, para poder “llamar” a la funcion

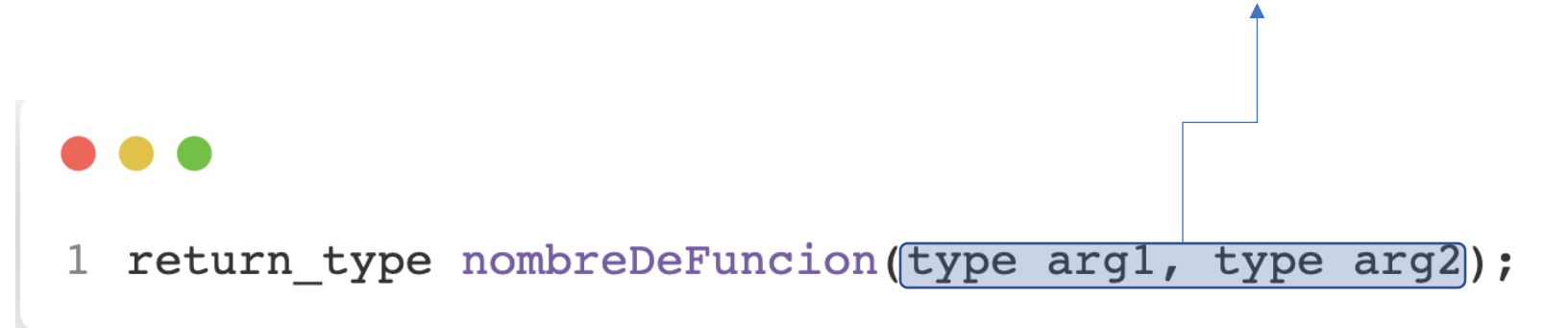


The diagram shows a function prototype enclosed in large square brackets. Above the prototype, a text box explains that the function name is its signature. A blue arrow points from this text to the function name in the code. To the left of the code, there are three colored circles (red, yellow, green) and a line number '1'.

```
1 return_type nombreDeFuncion(type arg1, type arg2);
```

Prototipo de la funcion

Argumentos o parámetros de la función.
Estos son algunos valores que la función
necesita para ser llamada



```
1 return_type nombreDeFuncion(type arg1, type arg2);
```

The diagram illustrates a function prototype. A blue arrow originates from the parameter list `(type arg1, type arg2)` in the code snippet and points upwards towards the text "Argumentos o parámetros de la función. Estos son algunos valores que la función necesita para ser llamada". The code snippet is enclosed in large square brackets, and the parameter list is highlighted with a light blue background.

Prototipo de la función: Ejemplo



```
1 int add_numbers(int a, int b);
```



```
1 string some_function(int a, string b);
```


Prototipo de la función: Void

- La función puede no devolver ningún resultado. Para ello se utiliza el tipo de dato void.
- Esta función no recibe ningún parámetro y no devuelve un resultado.



```
1 void other_function();
```

Cuerpo de la funcion

- El cuerpo de la función es la implementación. Es decir, la lista de instrucciones que debe ejecutar la función cada vez que esta es llamada.



```
1 int add_number(int a, int b) {  
2     int sum = a + b;  
3     cout << "Hello from here" << endl;  
4     cout << "Another line of code" << endl;  
5     // return the sum  
6     return sum;  
7 }
```

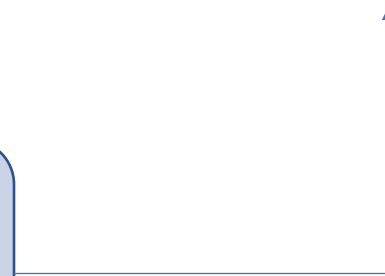
Cuerpo de la función

- El cuerpo de la función es la implementación. Es decir, la lista de instrucciones que debe ejecutar la función cada vez que esta es llamada.



```
1 int add_number(int a, int b) {  
2     int sum = a + b;  
3     cout << "Hello from here" << endl;  
4     cout << "Another line of code" << endl;  
5     // return the sum  
6     return sum;  
7 }
```

Cuerpo de la función. Instrucciones a realizar



- Esto se le llama declaración de una función.

Llamada de funciones

- Para llamar a la función basta con usar su nombre y pasarle los argumentos necesarios.
- Ya hemos estado utilizando algunas funciones, recuerdan?

Llamada de funciones

- Para llamar a la función basta con usar su nombre y pasarle los argumentos necesarios.
- Ya hemos estado utilizando algunas funciones, recuerdan?



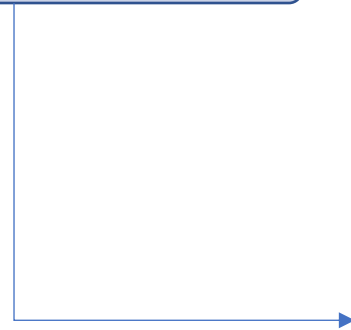
```
1 nombre_de_funcion(val1, val2);
```

Llamada de funciones

- Para llamar a la función basta con usar su nombre y pasarle los argumentos necesarios.
- Ya hemos estado utilizando algunas funciones, recuerdan?



```
1 nombre_de_funcion(val1, val2);
```



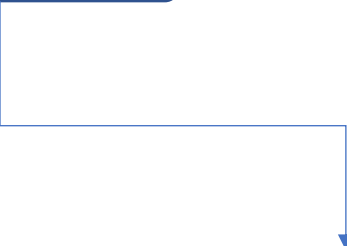
Nombre de la función que se quiere llamar.

Llamada de funciones

- Para llamar a la función basta con usar su nombre y pasarle los argumentos necesarios.
- Ya hemos estado utilizando algunas funciones, recuerdan?



```
1 nombre_de_funcion(val1, val2);
```



Valores que acepta la función. Estos deben ser del mismo tipo que se indica en el prototipo de la función.

Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```


Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```

Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```

Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```

Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Llamada de funciones



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Funciones

Tenemos que hacer tres cosas

- Crear el prototipo de la funcion
- Definir el cuerpo de la función (declaración)
- Llamar a la función

Nombre de nuestra funcion



Argumentos



```
1 #include <iostream>
2
3 using namespace std;
4
5 void print_welcome();
6
7 int main() {
8     cout << "Hello world!" << endl;
9 }
10
11 void print_welcome() {
12     cout << "Hello there from a function!:D" << endl;
13     cout << "Here we can write more lines" << endl;
14     cout << "other line D:" << endl;
15     cout << "so far so good" << endl;
16     cout << "again and again" << endl;
17 }
```

Argumentos



```
1 #include <iostream>
2
3 using namespace std;
4
5 void print_welcome();
6
7 int main() {
8     cout << "Hello world!" << endl;
9 }
10
11 void print_welcome() {
12     cout << "Hello there from a function!:D" << endl;
13     cout << "Here we can write more lines" << endl;
14     cout << "other line D:" << endl;
15     cout << "so far so good" << endl;
16     cout << "again and again" << endl;
17 }
```

No devuelve nada



Argumentos



```
1 #include <iostream>
2
3 using namespace std;
4
5 void print_welcome();
6
7 int main() {
8     cout << "Hello world!" << endl;
9 }
10
11 void print_welcome() {
12     cout << "Hello there from a function!:D" << endl;
13     cout << "Here we can write more lines" << endl;
14     cout << "other line D:" << endl;
15     cout << "so far so good" << endl;
16     cout << "again and again" << endl;
17 }
```

No recibe ningún argumento

Argumentos



```
1 #include <iostream>
2
3 using namespace std;
4
5 void print_welcome();
6
7 int main() {
8     cout << "Hello world!" << endl;
9     print_welcome();
10    print_welcome();
11    print_welcome();
12 }
13
14 void print_welcome() {
15     cout << "Hello there from a function!:D" << endl;
16     cout << "Here we can write more lines" << endl;
17     cout << "other line D:" << endl;
18     cout << "so far so good" << endl;
19     cout << "again and again" << endl;
20 }
```

→ Llamadas a la funcion

Argumentos



```
1 #include <iostream>
2
3 using namespace std;
4
5 int add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9 }
10
11 int add_numbers(int a, int b) {
12     int sum = a + b;
13     return sum;
14 }
```

Argumentos

Esta función debe
devolver un valor
entero



```
1 #include <iostream>
2
3 using namespace std;
4
5 int add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9 }
10
11 int add_numbers(int a, int b) {
12     int sum = a + b;
13     return sum;
14 }
```

Argumentos



```
1 #include <iostream>
2
3 using namespace std;
4
5 int add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9 }
10
11 int add_numbers(int a, int b) {
12     int sum = a + b;
13     return sum;
14 }
```

Recibe dos
argumentos de tipo
entero cada uno

Argumentos



```
1 #include <iostream>
2
3 using namespace std;
4
5 int add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9 }
10
11 int add_numbers(int a, int b) {
12     int sum = a + b;
13     return sum;
14 }
```

Cuerpo de la funcion

Argumentos



```
1 #include <iostream>
2
3 using namespace std;
4
5 int add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9 }
10
11 int add_numbers(int a, int b) {
12     int sum = a + b;
13     return sum;
14 }
```

Retorno de la funcion

Argumentos



```
1  #include <iostream>
2
3  using namespace std;
4
5  int add_numbers(int a, int b);
6
7  int main() {
8      cout << "Hello world!" << endl;
9      cout << add_numbers(5, 5) << endl;
10 }
11
12 int add_numbers(int a, int b) {
13     int sum = a + b;
14     return sum;
15 }
```

Llamada a la función con dos valores enteros.

Argumentos

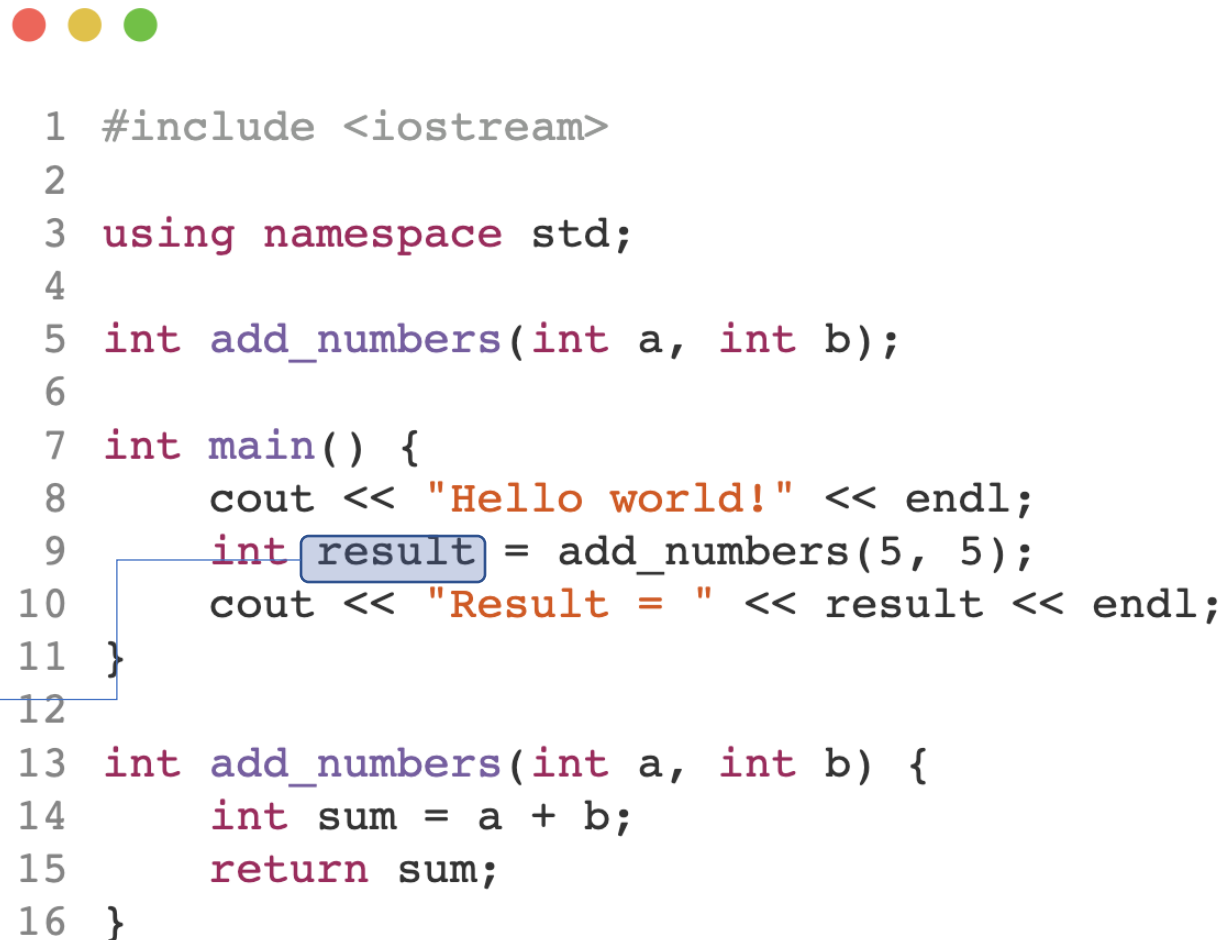


```
1 #include <iostream>
2
3 using namespace std;
4
5 int add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9     int result = add_numbers(5, 5);
10    cout << "Result = " << result << endl;
11 }
12
13 int add_numbers(int a, int b) {
14     int sum = a + b;
15     return sum;
16 }
```

Llamada a la función.
Esta función retorna la
suma de sus
argumentos.

Argumentos

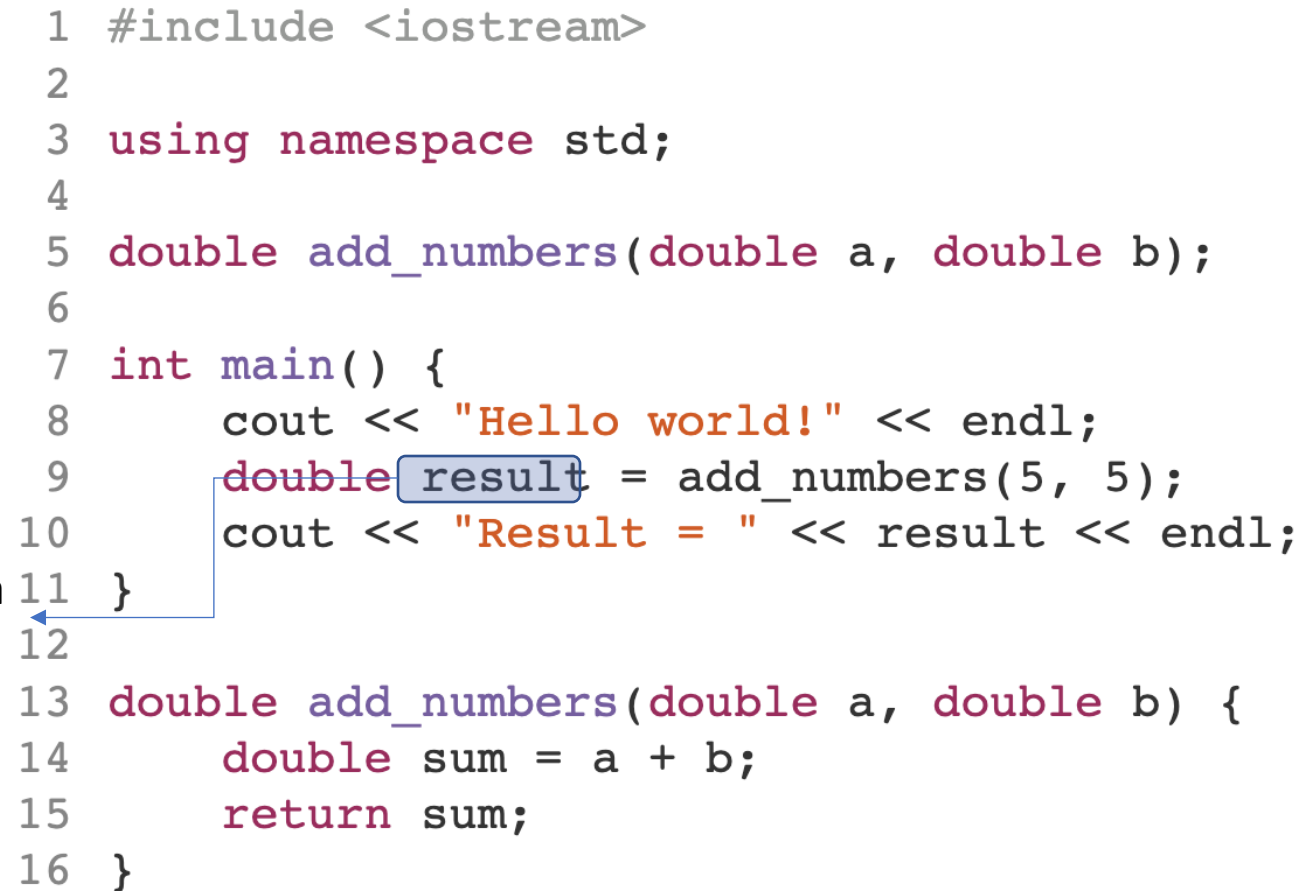
La devolución de la función se puede guardar en una variable. Note que esta variable tiene que ser el mismo tipo de dato que lo que retorna la función



```
1 #include <iostream>
2
3 using namespace std;
4
5 int add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9     int result = add_numbers(5, 5);
10    cout << "Result = " << result << endl;
11 }
12
13 int add_numbers(int a, int b) {
14     int sum = a + b;
15     return sum;
16 }
```



Argumentos

La devolución de la función se puede guardar en una variable. Note que esta variable tiene que ser el mismo tipo de dato que lo que retorna la función



```
1 #include <iostream>
2
3 using namespace std;
4
5 double add_numbers(double a, double b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9     double result = add_numbers(5, 5);
10    cout << "Result = " << result << endl;
11 }
12
13 double add_numbers(double a, double b) {
14     double sum = a + b;
15     return sum;
16 }
```

Argumentos



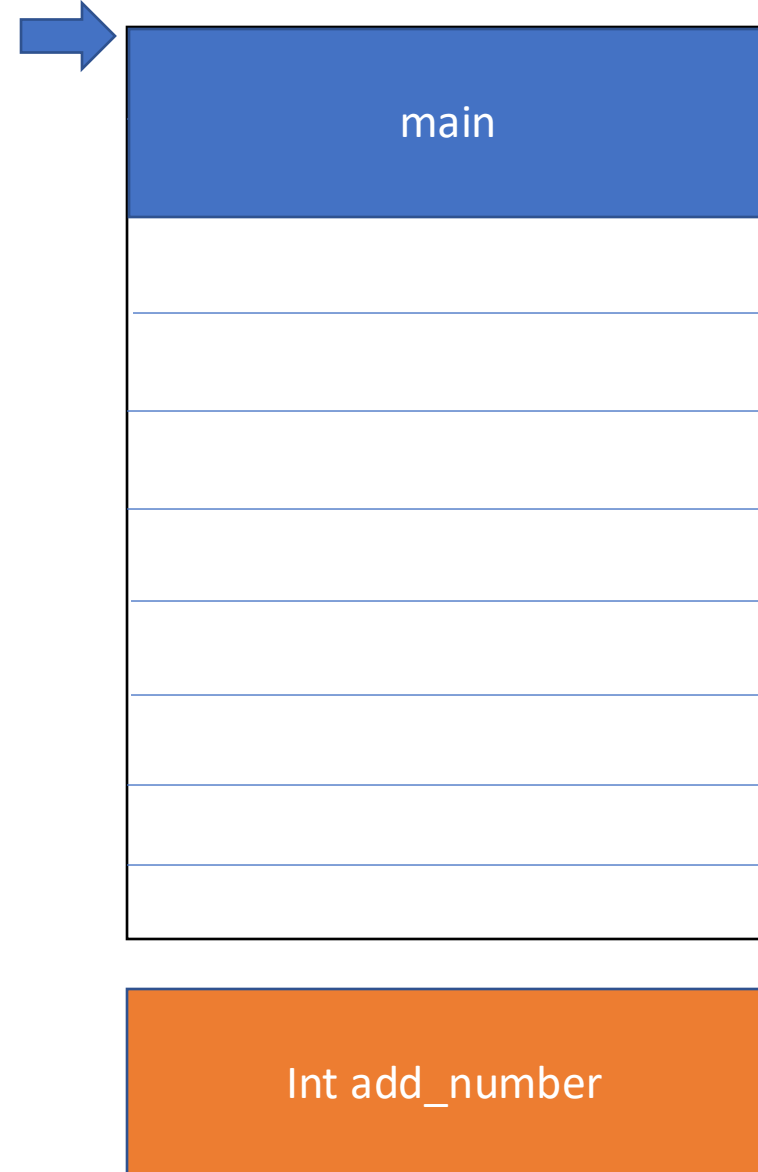
```
1 #include <iostream>
2
3 using namespace std;
4
5 void print_string(string str);
6
7 int main() {
8     cout << "Hello world!" << endl;
9
10    print_string("My new string");
11
12    cout << "Result = " << result << endl;
13 }
14
15 void print_string(string str) {
16     cout << "The string is " << str << endl;
17 }
```

Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```

Memoria RAM



Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```

Memoria RAM

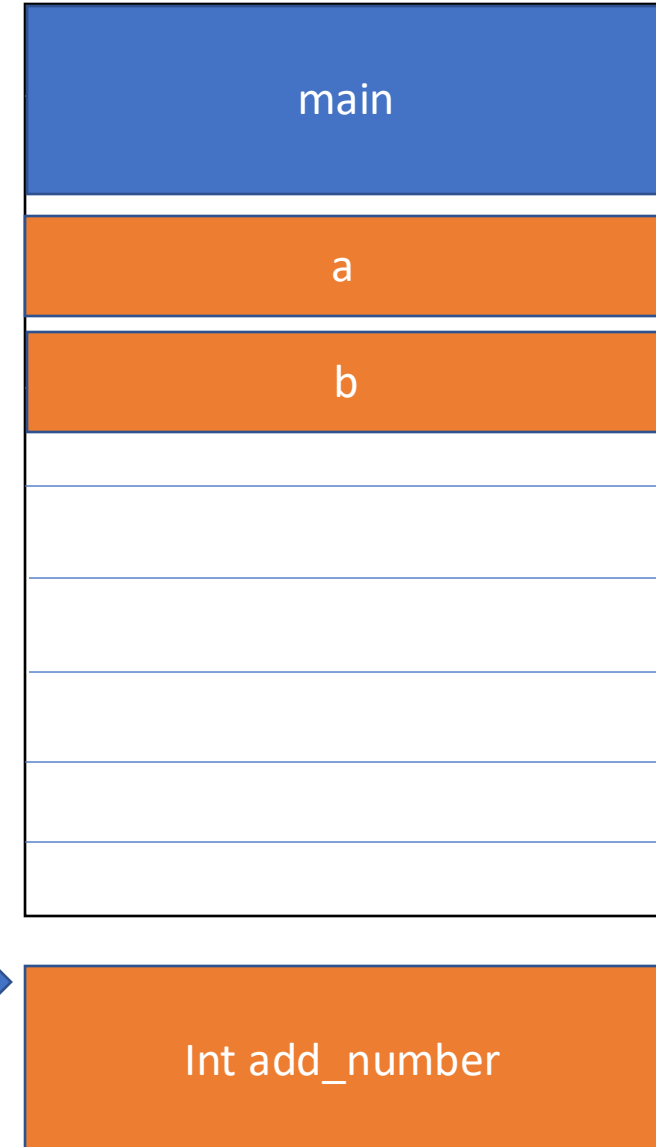


Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```

Memoria RAM



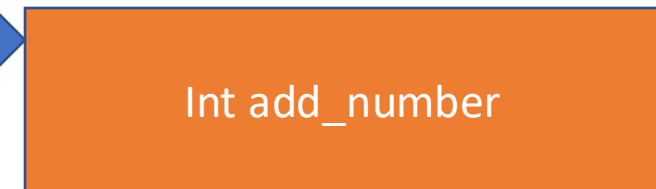
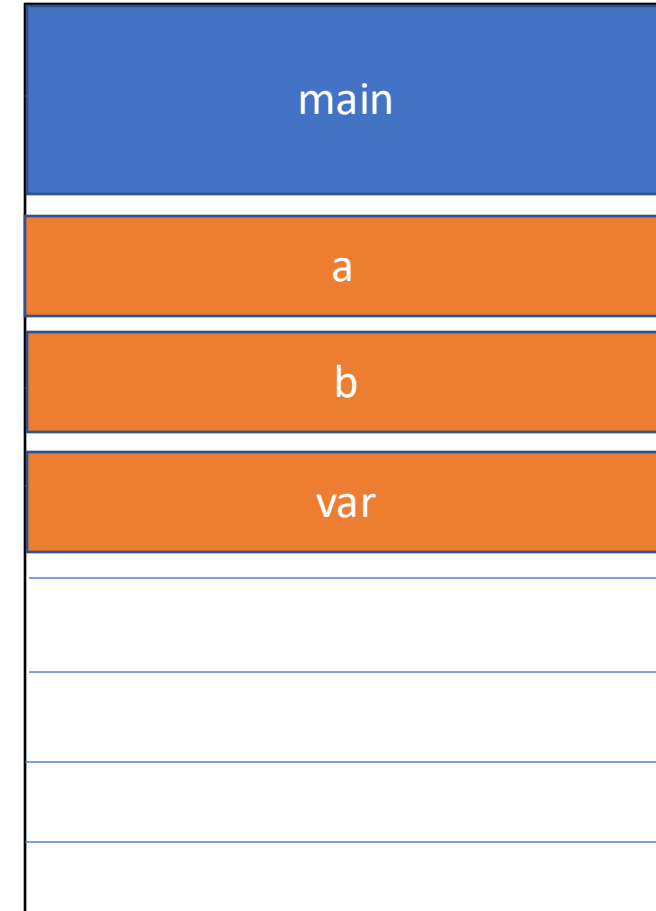
Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM



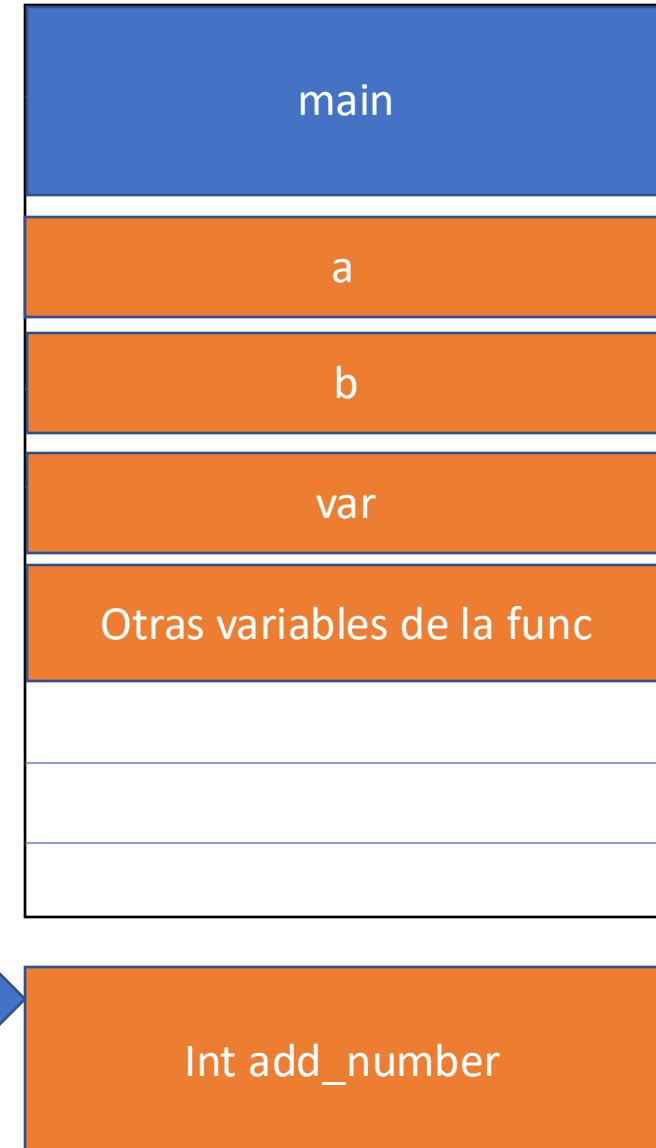
Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM



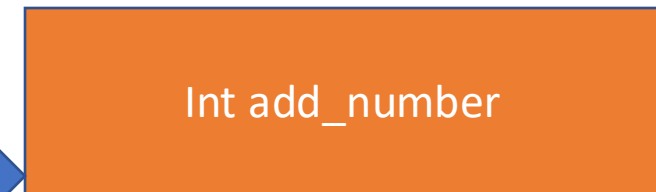
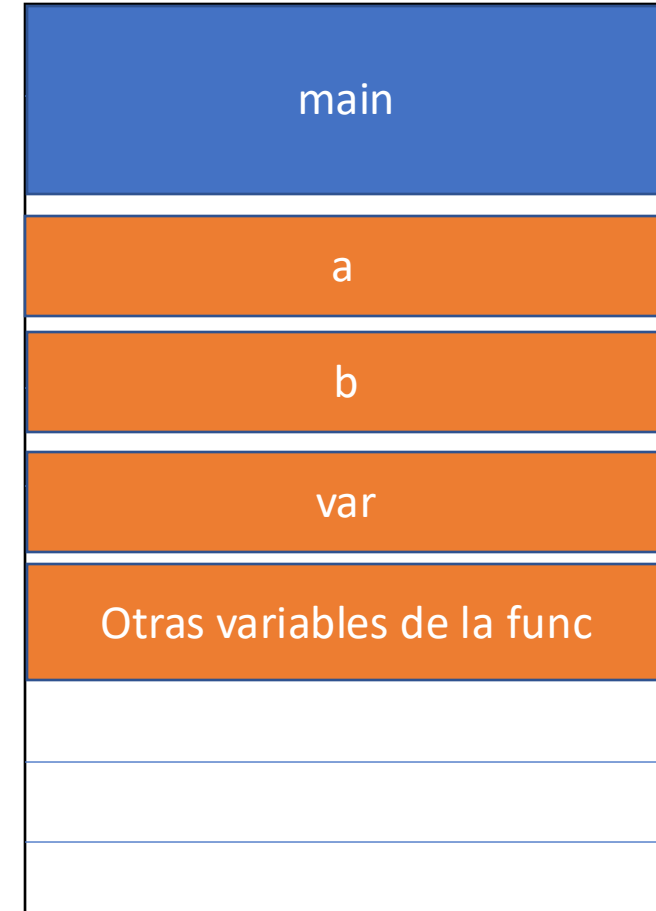
Memoria




```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM



Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```

Memoria RAM



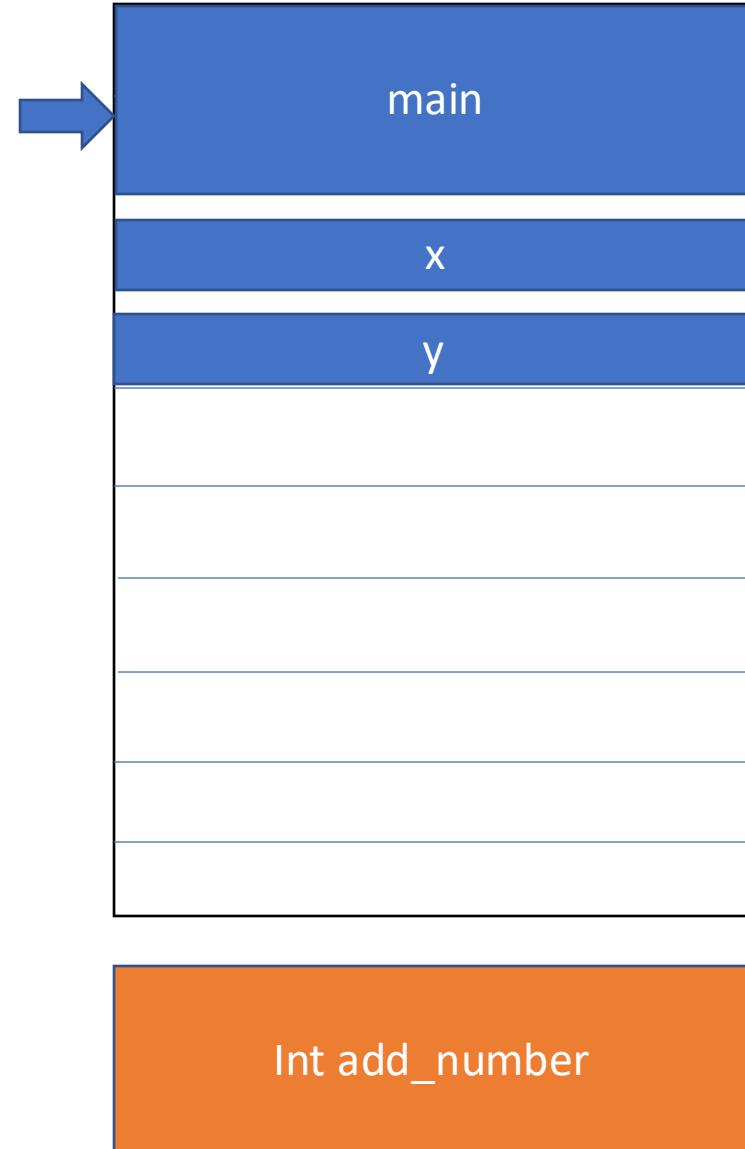
Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM



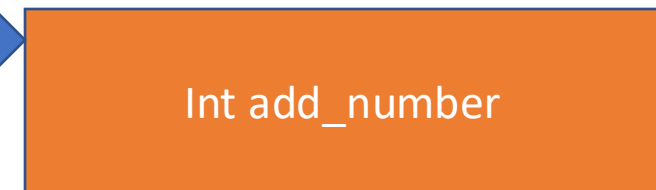
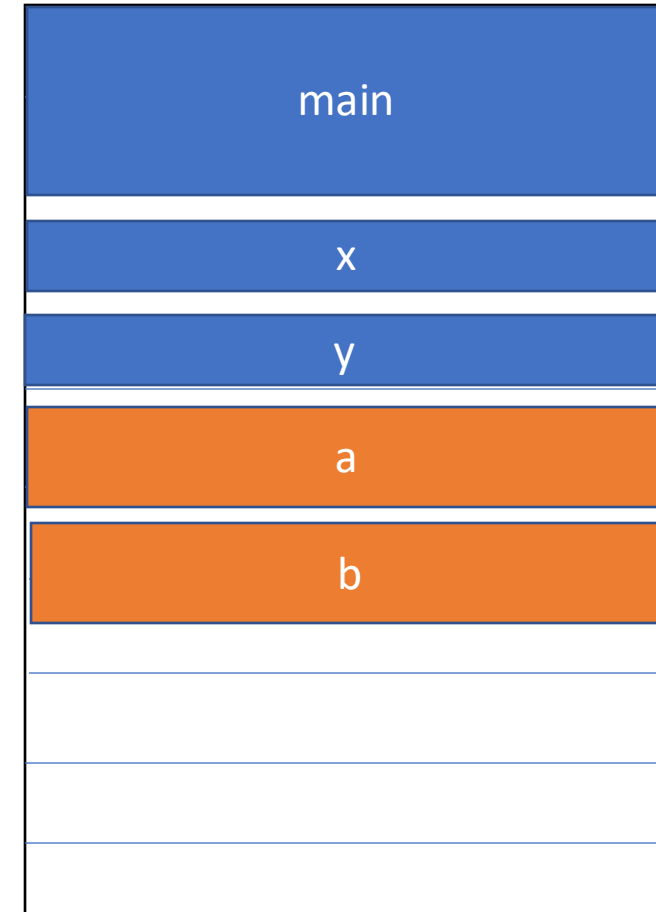
Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM



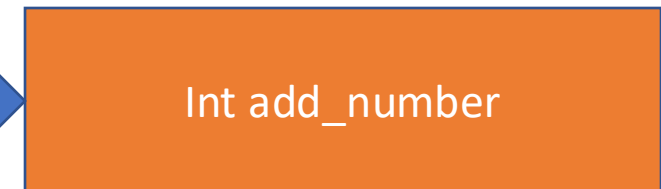
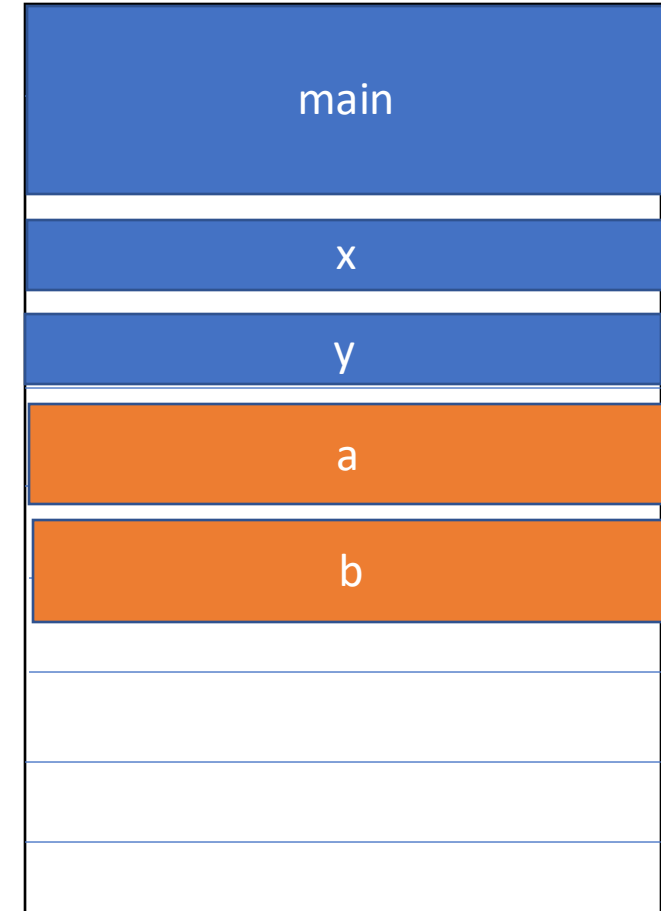
Memoria



```
1  int add_number(int a, int b);
2
3  int main() {
4      cout << "From main!" << endl;
5
6      // llamada a la funcion
7      // se le pasa dos valores int
8      cout << add_number(4, 4);
9
10     // tambien se le puede pasar variables
11     // de tipo int
12     int x = 10;
13     int x = 11;
14     cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM



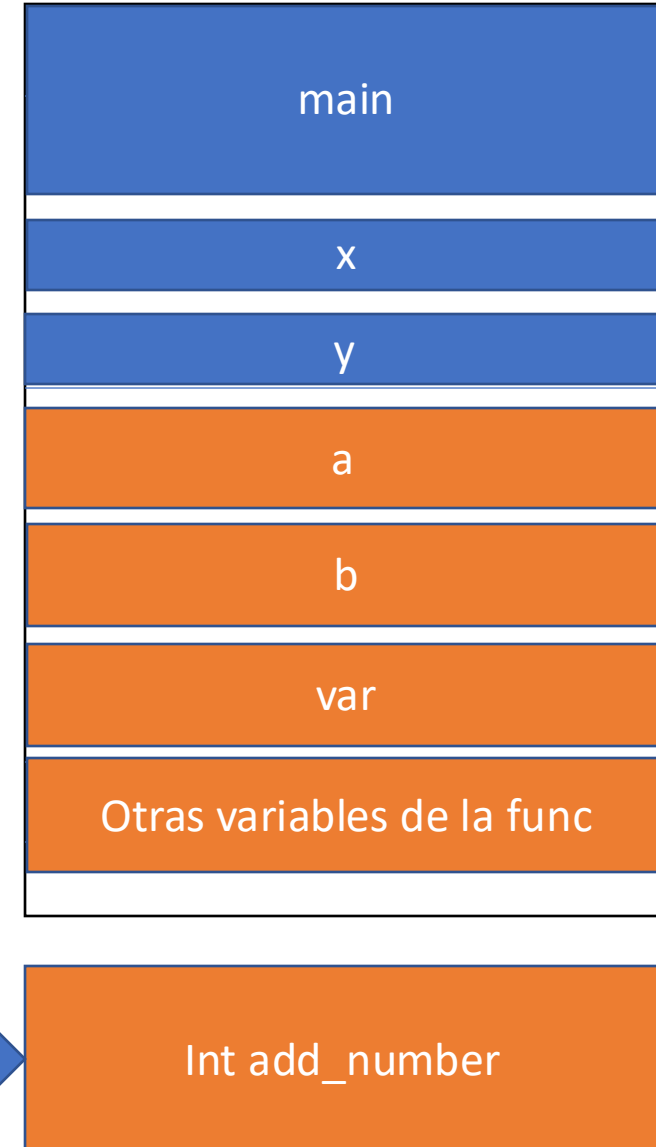
Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM



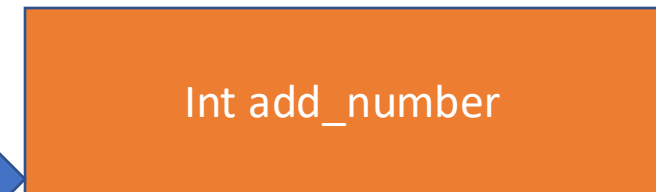
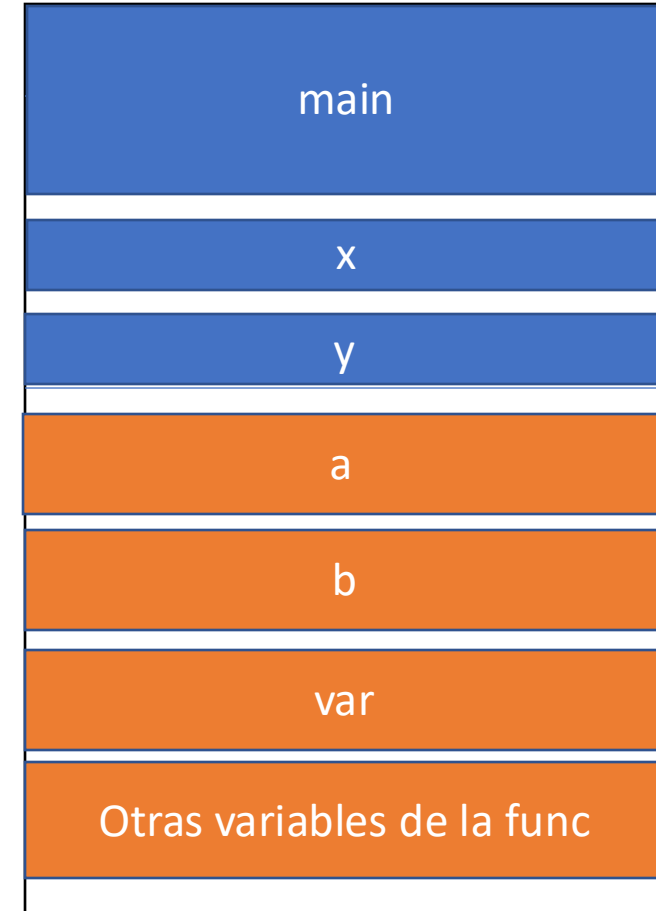
Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM

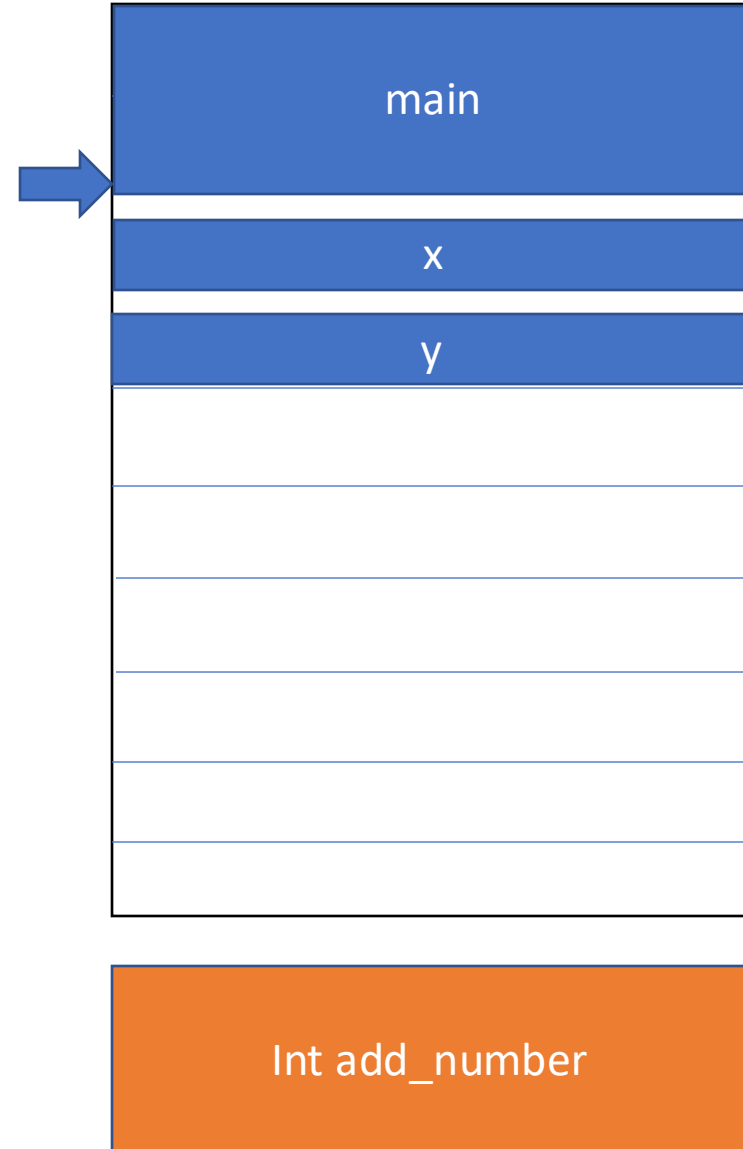


Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```

Memoria RAM



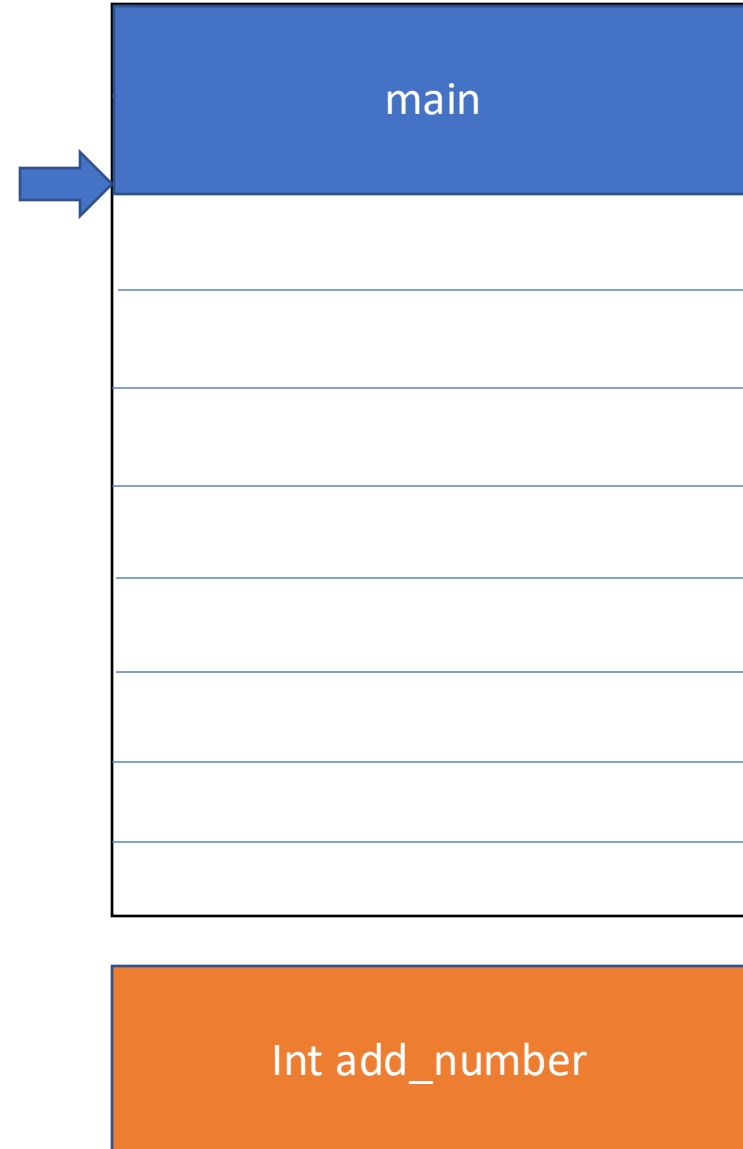
Memoria



```
1 int add_number(int a, int b);
2
3 int main() {
4     cout << "From main!" << endl;
5
6     // llamada a la funcion
7     // se le pasa dos valores int
8     cout << add_number(4, 4);
9
10    // tambien se le puede pasar variables
11    // de tipo int
12    int x = 10;
13    int x = 11;
14    cout << add_number(x, y);
15 }
16
17 int add_number(int a, int b) {
18     int var = 0;
19     // some code
20
21     // end of code
22     return var;
23 }
```



Memoria RAM



Scope (alcance)

La variable sum
únicamente existe en
la función
(add_numbers)
Afuera de esta función
ya no existe

```
1 #include <iostream>
2
3 using namespace std;
4
5 double add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9
10    double result = add_numbers(1, 2);
11
12    cout << "Result = " << result << endl;
13
14    // No podemos acceder a variables internas de
15    // la funcion. Por ejemplo, sum unicamente existe en
16    // la funcion add_numbers. La siguiente linea devolvera
17    // un error:
18    // cout << sum << endl;
19
20    // Incluso los argumentos: (la siguiente linea es un error)
21    // cout << a;
22    // cout << b;
23 }
24
25 double add_numbers(int a, int b) {
26     double sum = a + b;
27     return sum;
28 }
```

Scope (alcance)



```
1 #include <iostream>
2
3 int globalVar = 10; // global variable
4
5 int main() {
6     int localVar = 5; // local variable
7     std::cout << "Local variable value: " << localVar << std::endl;
8     std::cout << "Global variable value: " << globalVar << std::endl;
9     return 0;
10 }
```

Variable global – es una variable que se puede acceder y modificar desde cualquier parte del programa

Scope (alcance)

- En programación, el alcance se refiere a la región del código donde una variable, función u otra entidad de programación es visible y accesible.
- El alcance de una variable determina dónde se puede acceder o modificar dentro del programa.
- El ámbito de una variable está determinado por el lugar en el que se declara en el código.
- Las variables declaradas dentro de una función o bloque de código tienen un alcance local y solo son visibles dentro de esa función o bloque.
- Las variables declaradas fuera de cualquier función tienen un alcance global y son visibles para todas las funciones del programa.

```
1 #include <iostream>
2
3 using namespace std;
4
5 double add_numbers(int a, int b);
6
7 int main() {
8     cout << "Hello world!" << endl;
9
10    double result = add_numbers(1, 2);
11
12    cout << "Result = " << result << endl;
13
14    // No podemos acceder a variables internas de
15    // la funcion. Por ejemplo, sum unicamente existe en
16    // la funcion add_numbers. La siguiente linea devolvera
17    // un error:
18    // cout << sum << endl;
19
20    // Incluso los argumentos: (la siguiente linea es un error)
21    // cout << a;
22    // cout << b;
23 }
24
25 double add_numbers(int a, int b) {
26     double sum = a + b;
27     return sum;
28 }
```