# CHAPTER-2

Basic of c++ programming

## Introduction to c++ programming language

- ✓ C++ is an object-oriented programming language.
- ✓ It was developed by Bjarne Stroustrup (pronounced BYAR-neh) at AT&T Bell Laboratories in the early 1980s.
- ✓ The goal was to combine the strengths of two programming approaches –
  - *Support for object-oriented programming*
  - *Retain the power and efficiency of C*
- ✓ The power of C includes Speed, Efficiency , Low memory usage
- ✓ This combination led to the development of the C++ programming language.
- ✓ C++ is a versatile language for handling very large program.
- ✓ C++ is suitable for developing games, compilers, and other complex real-life application systems

## Interview Question??

### Why C++ and not ++C?

- The name **C++** comes from the **++** symbol in C, which means "add one."
- **C++** means it is the next better version of **C**.
- **++C** means something different in programming (it means "increment C before using it"), so it was not used as the name.
- The name **C++** shows that it is an improved version of **C**.

Prepared by Ayush Lamsal | |
layush788@gmail.com

## Structure of c++

The c++ Program is written using a specific template structure . The structure of the program written in c++ language is divided into the following sections:

| |
|---|
| Document section in c++ program |
| Linking section |
| Definition section |
| Global declaration section: |
| Class Definition |
| Main function section |

Fig: Structure of c++

## Structure of c++ program

**Document section in c++ program**

- The document section comes first in a c++ program
- It is used to document the logic or purpose of the program
- You can also write a notes for programmer or user to understand what the program does
- Everything written in this section is treated as a comment and is not compiled or executed.
- This helps to improve readability and maintainability of the code.
- This section is optional  which means the program can run without it

## Example of document section

```
// Program to calculate the area of a circle
// Written by: Ayush Lamsal
// Date: 10-May-2025
```

Note: This is written at the beginning of a program and  improves the readability of a program

## Structure of c++ program

**Linking section:**

Linking section is a compulsory section in the structure of c++ contains two parts:

1. Header file:

Header file contains predefined classes, functions, constants, etc.

They are part of the C++ Standard Library and were defined during the development of C++. They allow the programmer to use built-in functions, classes, and constants to simplify coding and save time.

They are included  in c++ program using the #include directive.

Allows the program to use the methods and attributes defined in that file.

For example

#include <Iostream>

<iostream> is a header file in C++.

By including the <iostream> header file, we can use various classes, methods, functions, and objects defined inside it.

It provides important objects like cin (for input) and cout (for output

## Structure of c++ program

**Linking section:**

Linking section  in the structure of c++ contains two parts:

2. namespace:

- *Namespace is another part of the linking section.*
- *It is used for grouping of entities like class , object, function etc. under a single name*
- *It helps define which namespace an object belongs to, helping to avoid naming conflicts in a program.*

Namespaces can be accessed in multiple ways:

- using namespace std;
- using std :: cout;

## Structure of c++ program

**Definition section:**

The Definition Section is an optional part of a C++ program's structure; the program can run even without it.

In this section, we define the values of some constants using directives like #define.

For example, #define PI 3.14159 tells the preprocessor to replace every occurrence of PI in the code with 3.14159 before compilation.

Also, in the Definition Section, we provide the full implementation (body) of functions that were declared earlier (Generally from main method). This means writing the actual code inside the functions so they can perform their tasks when called in the program.

## Structure of c++ program

**Global declaration section:**

- This section is used to declare global variables, constants, and functions that can be accessed throughout the entire program.
- Global declarations are made outside of all functions, usually at the top of the program.
- Variables declared here can be used by any function within the program.
- Helps to share data between different parts of the program without passing them as function parameters.

## Structure of c++ program

**Class Definition**

In this section we declare the class using the class keyword .

This section is also a optional as we can execute the c++ program even without a class.

In this section, we define the class along with its attributes (data members) and methods (member functions) that the class contains.

**Main function section**

- It is a compulsory function in a c++
- The main function tells the compiler where to start the execution of the program. The execution of the program starts with the main function.
- All the statements that are to be executed are written in the main function.
- The compiler executes all the instructions which are written in the curly braces **{}** which encloses the body of the main function.

## Input and output Operators

### Input operators:

- For input operators we used scanf() in c but in c++ we use cin >> instead of scanf().
- In scanf() we used double quotation(“ “) and address (&) operator before variable in whichi data is to be stored but we don’t use both of them in c++.

For example:

- *Cin >> number1;*
- *In this number1 is variable in which data is stored.*

### Output operator:

- For output operator in c we used prinft() but in c++ we use cout << and used as follows in c++ programming
  - *cout < “statement” << printing variable;*

## Input and output Operators

### Character set and Tokens:

A **Character Set** is a collection of all the characters that a program can understand, and we can use them in the program. These include alphabets, digits, special symbols, and whitespace.

Some of the characters included in c++ are:

| Types of characters | Examples |
|---|---|
| Alphabets | A–Z (uppercase), a–z (lowercase) |
| Digits | 0, 1, 2, 3, ..., 9 |
| Special characters | +, -, *, /, =, ;, { }, ( ), %, @, #, etc. |
| whitespace | Space, tab, newline |

Example
int age = 25;  // Uses characters: i, n, t, a, g, e, =, 2, 5, ;

Prepared by Ayush Lamsal | |
layush788@gmail.com

## Tokens

A token is a smallest meaningful unit  in a program. It is made by combing a character from the character set and has a specific meaning in  the code.

For example:

int is a token because it is the smallest meaningful unit in the program. It specifies that a variable is of integer type.

When the compiler reads your codes it break down into the token to understand and process the program

These are the building block In a c++ language or any other language which are constructed together to write a program. These are of 6 types which are as follows:

## Tokens

Keywords:

- **Keywords** are certain reserved words in C++ that have predefined meanings and specific purposes.
- These words are **reserved** and cannot be used as names for variables, functions, or identifiers.
- Keywords are **always written in lowercase**.
- **Standard C++** has **48 keywords**.
- **ANSI** (American National Standards Institute) traditionally added **15 more keywords** when standardizing the C++ programming language for the first time.
- **Standardization** means writing a program that works the same way on different computers and compilers.
- Later, the C++ language mostly uses the **48 standard keywords**, and the additional **15 keywords introduced by ANSI** are rarely used or have become obsolete in modern development.
- Some example of keyword is:
  - *Int, float , return, else etc.*

## Tokens

Identifiers:

All the words used in a C++ program are either keywords or identifiers.

Keywords are predefined by the language and cannot be changed by the user.

Identifiers are user-defined names used to give names to various entities like variables, arrays, functions, and structures.

## Tokens

Identifiers:

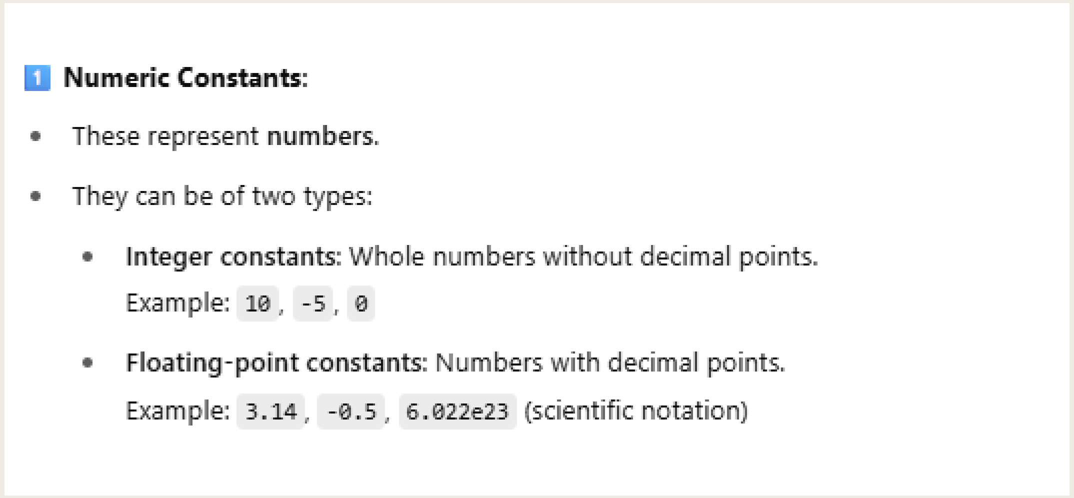
All the words used in a C++ program are either keywords or identifiers.

Keywords are predefined by the language and cannot be changed by the user.

Identifiers are user-defined names used to give names to various entities like variables, arrays, functions, and structures

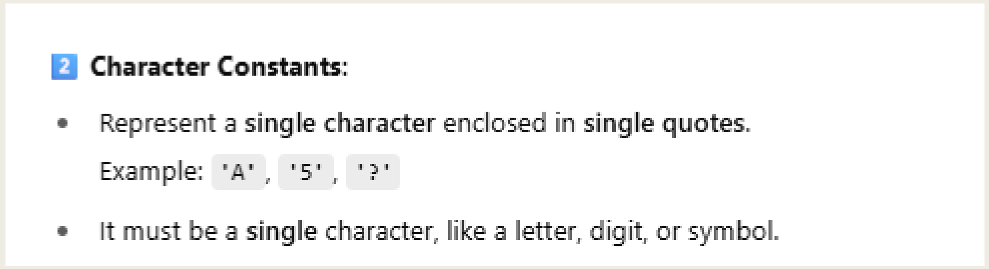
## Tokens

- A constant is a value that cannot be changed during the execution of the program.
- These fixed values are also called literals.
- Constant can be of three types:



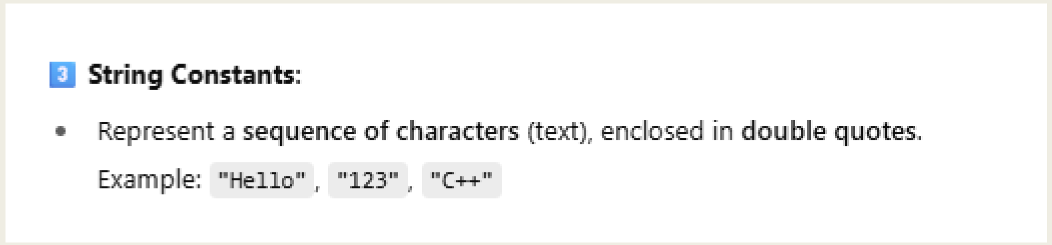
1 **Numeric Constants:**

- These represent **numbers.**
- They can be of two types:
  - **Integer constants:** Whole numbers without decimal points.
    Example: `10`, `-5`, `0`
  - **Floating-point constants:** Numbers with decimal points.
    Example: `3.14`, `-0.5`, `6.022e23` (scientific notation)



2 **Character Constants:**

- Represent a **single character** enclosed in **single quotes.**
  Example: `'A'`, `'5'`, `'?'`
- It must be a **single** character, like a letter, digit, or symbol.



3 **String Constants:**

- Represent a **sequence of characters** (text), enclosed in **double quotes.**
  Example: `"Hello"`, `"123"`, `"C++"`

- If we need to use a constant many times in a program, we can give it a name.
- For example, if we need to use the constant 3.1415 many times, we can define a symbolic or named constant like this:
  - const float pi = 3.1415;

  Now, instead of writing 3.1415 everywhere, we can use pi.

## Tokens

- In C++, we can also define constants using preprocessor directives like #define.
- This is called a macro constant or also called as symbolic constant.

```
#define PI 3.1425
```

- When we define constants using #define (a preprocessor directive), no data type is specified.
- The preprocessor simply replaces the name with the value before compilation.

## String

A string in a c++ is considered as a token

Since a string constant (like "Hello") is a fixed sequence of characters, it qualifies as a token.

- A string is an array of characters (or alphabets) stored together.
- In C++, a string is written inside double quotes (" ").
- Example: "Hello", "123", "C++“
- A string can contain:
  - *Alphabets (e.g., "Hello")*
  - *Digits (e.g., "1234")*
  - *Special characters (e.g., "@2024$")*
  - *Or a combination of all.*
- A **string constant** is also a **type of token** in C++.

## Operators

Operators are symbols that perform operations on variables and values.

Since tokens are the smallest meaningful units in a program, operators qualify as tokens too.

C++ has a rich set of operators.

All c operators are valid in c++ . In addition c++ introduce some new operator also and these are:

| | |
|---|---|
| << | Insertion operator |
| >> | Extraction operator |
| : : | Scope Resolution operator |
| endl: | Line feed operator |
| new: | memory allocated operator |
| delete: | memory release operator |

## Special symbols:

There are some special character or symbols in c++ such as  (), {}, ; , / etc

## Data types

In c++ data types are broadly classified into three categories

- *user-defined data –type*
- *basic data type*
- *derived data type*

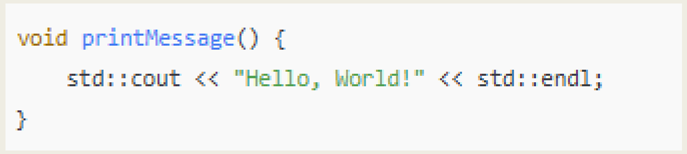
## Data types

### Built in data type:

- Built-in data types are also called predefined data types in C++, provided by the developer (C++ language).
- Built-in data types in C++ are char, int, float, double; the ANSI C++ committee added two more: bool (for logical values) and wchar_t (for wide characters)
- Note: A wide character means a character that is not a regular English (ASCII) character, but a character from other languages, such as Japanese, Hindi, Chinese, etc., that require more than 1 byte to store. wchar_t ch = L’अ’ (L before character indicates a wide character). It is not displayed in console so we don’t read about this here

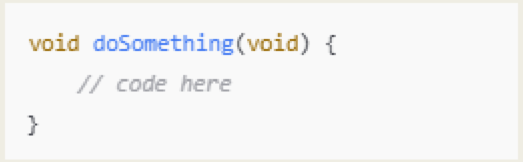
## Data types

- Normal use of void in c++ are:
- To specify the return type of function when it is not returning any value.



```cpp
void printMessage() {
    std::cout << "Hello, World!" << std::endl;
}
```

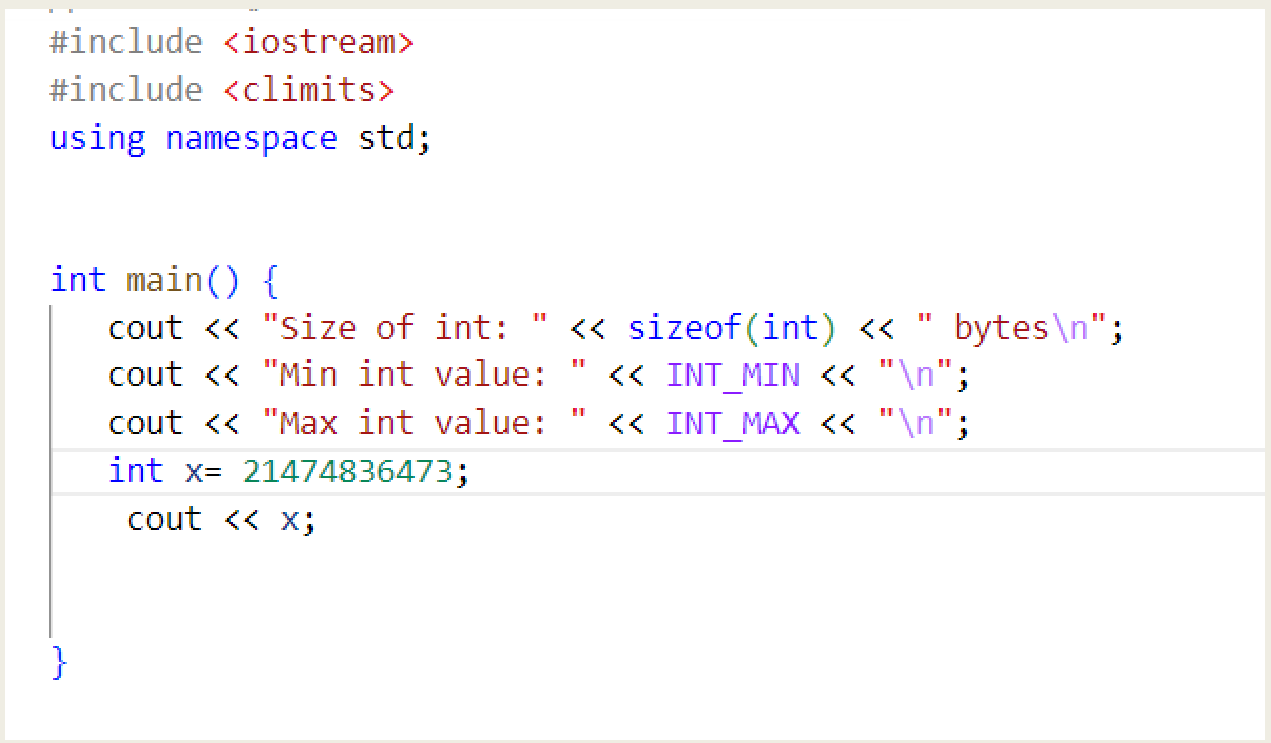
- To indicate an empty argument list to function



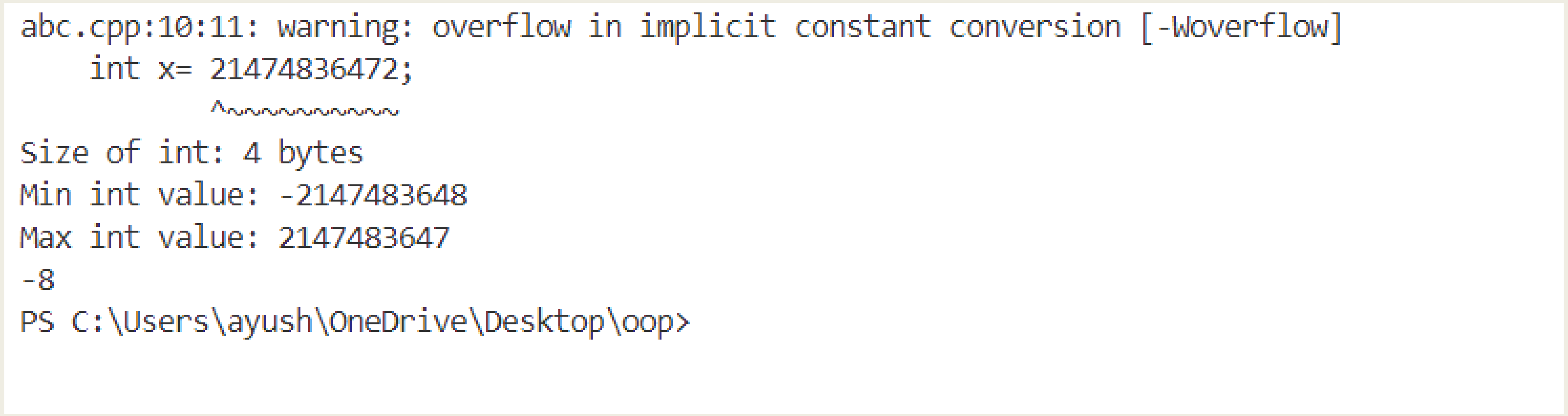
```cpp
void doSomething(void) {
    // code here
}
```

*Using void inside the parentheses means the function takes no arguments. In C++, just empty parentheses () also mean no parameters, so void doSomething() is equivalent.*

## Example of Range of int and its out of range example



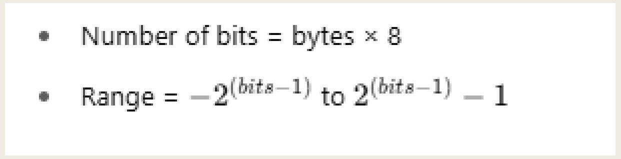
```cpp
#include <iostream>
#include <climits>
using namespace std;


int main() {
    cout << "Size of int: " << sizeof(int) << " bytes\n";
    cout << "Min int value: " << INT_MIN << "\n";
    cout << "Max int value: " << INT_MAX << "\n";
    int x= 21474836473;
     cout << x;


}
```



```
abc.cpp:10:11: warning: overflow in implicit constant conversion [-Woverflow]
     int x= 21474836472;
            ^~~~~~~~~~~
Size of int: 4 bytes
Min int value: -2147483648
Max int value: 2147483647
-8
PS C:\Users\ayush\OneDrive\Desktop\oop>
```

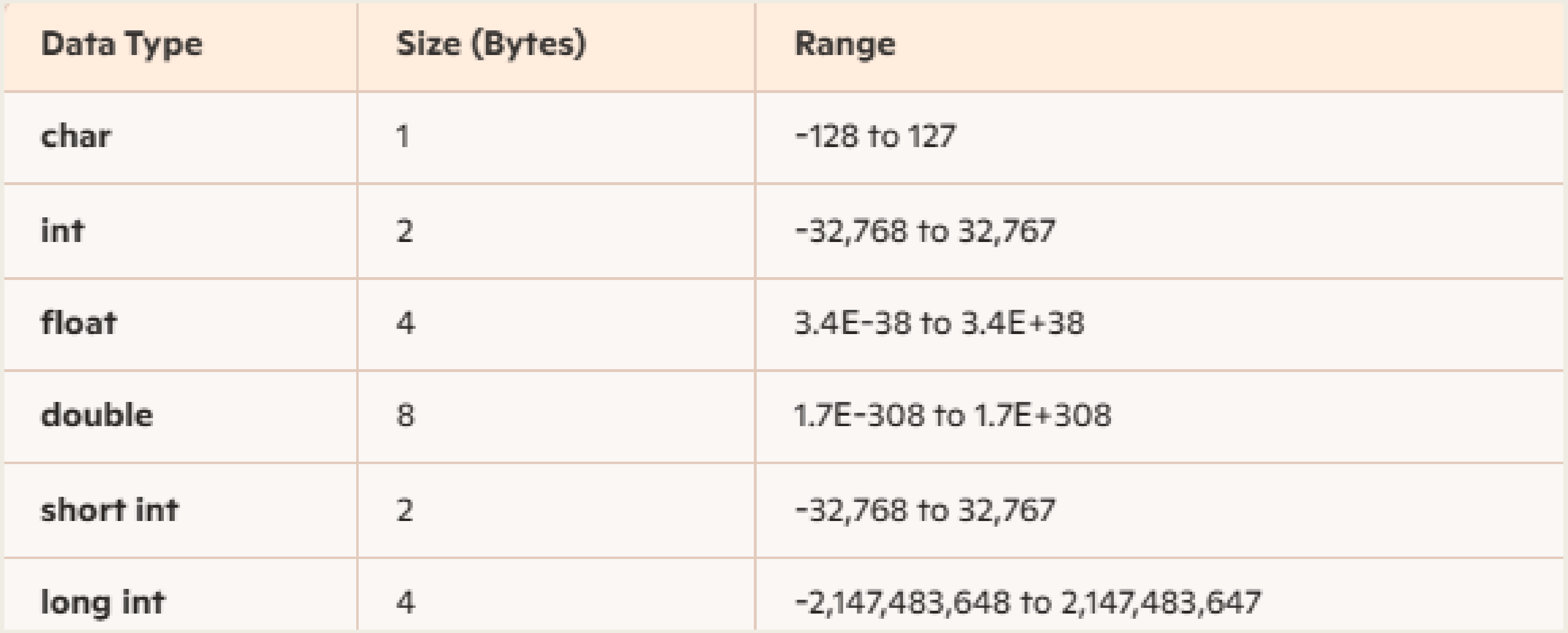
## Data types

Range is calculated using formula as:



- Number of bits = bytes × 8
- Range = $-2^{(bits-1)}$ to $2^{(bits-1)} - 1$

Size and range of some c++ basic data types are.



| Data Type | Size (Bytes) | Range |
|---|---|---|
| char | 1 | -128 to 127 |
| int | 2 | -32,768 to 32,767 |
| float | 4 | 3.4E-38 to 3.4E+38 |
| double | 8 | 1.7E-308 to 1.7E+308 |
| short int | 2 | -32,768 to 32,767 |
| long int | 4 | -2,147,483,648 to 2,147,483,647 |

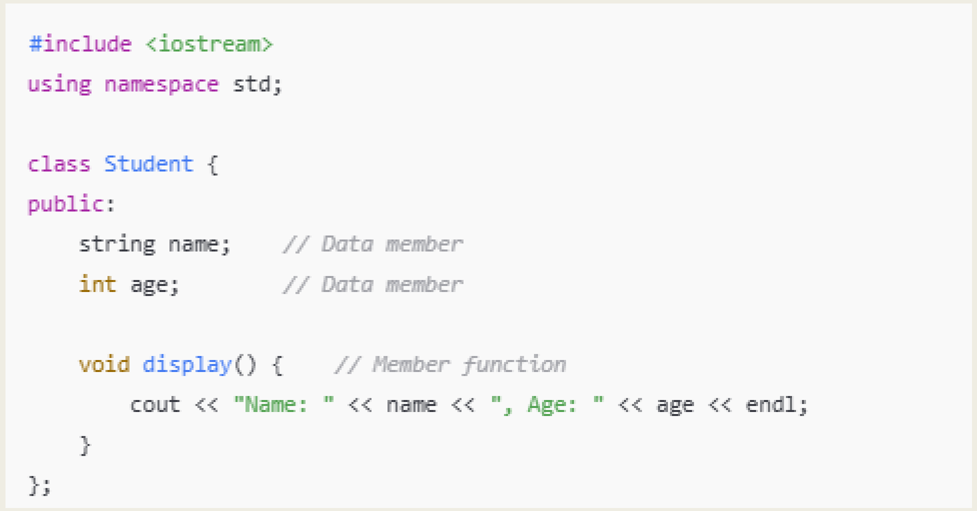
Note:
3.4E-38 means $3.4 \times 10^{-38}$ (a very small number).
1.7E+308 means $1.7 \times 10^{308}$ (a very large number).

## Data types

User Defined data type:

- In C, user-defined data types such as struct and union are commonly used to group related data together. These data types are also legal in C++, but C++ extends their capabilities by adding features that support object-oriented programming (OOP).
- One of the most important additions in C++ is the class keyword, which allows programmers to define their own data types with both data members and member functions (functions that operate on the data). A class can be used just like any other data type, and the variables created from a class are called objects.
- Objects are the central focus of object-oriented programming, which is based on the concept of encapsulating data and functionality together within classes.

## Data types



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

class Student {
public:
    string name;    // Data member
    int age;        // Data member

    void display() {    // Member function
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
```

- ➢ In the given above example Student is a user defined data type.
- ➢ It groups together data members (name, age) and member functions (display).
- ➢ You can create variables of type Student (called objects), just like you create variables of built-in types like int or char and we can store any value in it like char string as we want

## Data types

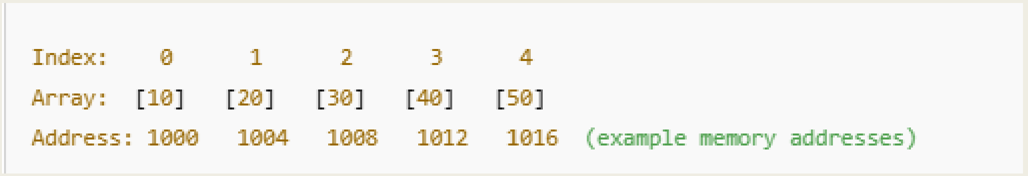
Derived data type:

A derived data type is a data type built from existing data type.

It is not a basic or a primitive data type itself but  created by combining , modifying or extending existing types

Some example of  Derived data types are:

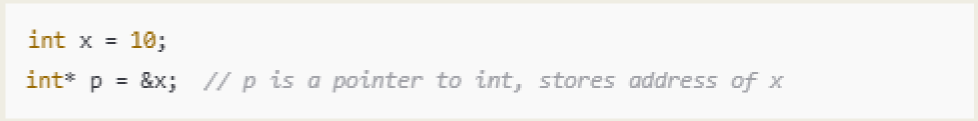
**Array:**



```
Index:     0       1       2       3       4
Array:  [10]    [20]    [30]    [40]    [50]
Address: 1000    1004    1008    1012    1016   (example memory addresses)
```

An array is a derived data type that stores multiple values of the same data type in a contiguous block of memory.
It allows you to store and access a collection of elements using a single variable name and an index.

**Pointer:**



```
int x = 10;
int* p = &x;  // p is a pointer to int, stores address of x
```

Pointer is a built form a built in type and stores the address of variable of that type

It derives form the base type but behave differently because it holds the memory address instead of direct values

## Type Conversion  (Important)

- **Type conversion**, also called **type casting**, is the process of changing a value from one data type to another.
- Sometimes, you need to work with different data types together—for example, converting a floating-point number to an integer or vice versa. Type conversion helps the program handle these situations safely by converting values to the appropriate data type.

**Real life example**

**You are creating a billing system for a grocery store.**

- The prices of items are stored as **floating-point numbers** (like 5.99 dollars).
- But the cash register only accepts **whole dollar amounts** (integers).

Before sending the total price to the cash register, you need to **convert the floating-point total to an integer** by dropping the cents.

Type conversion can be done in two ways:

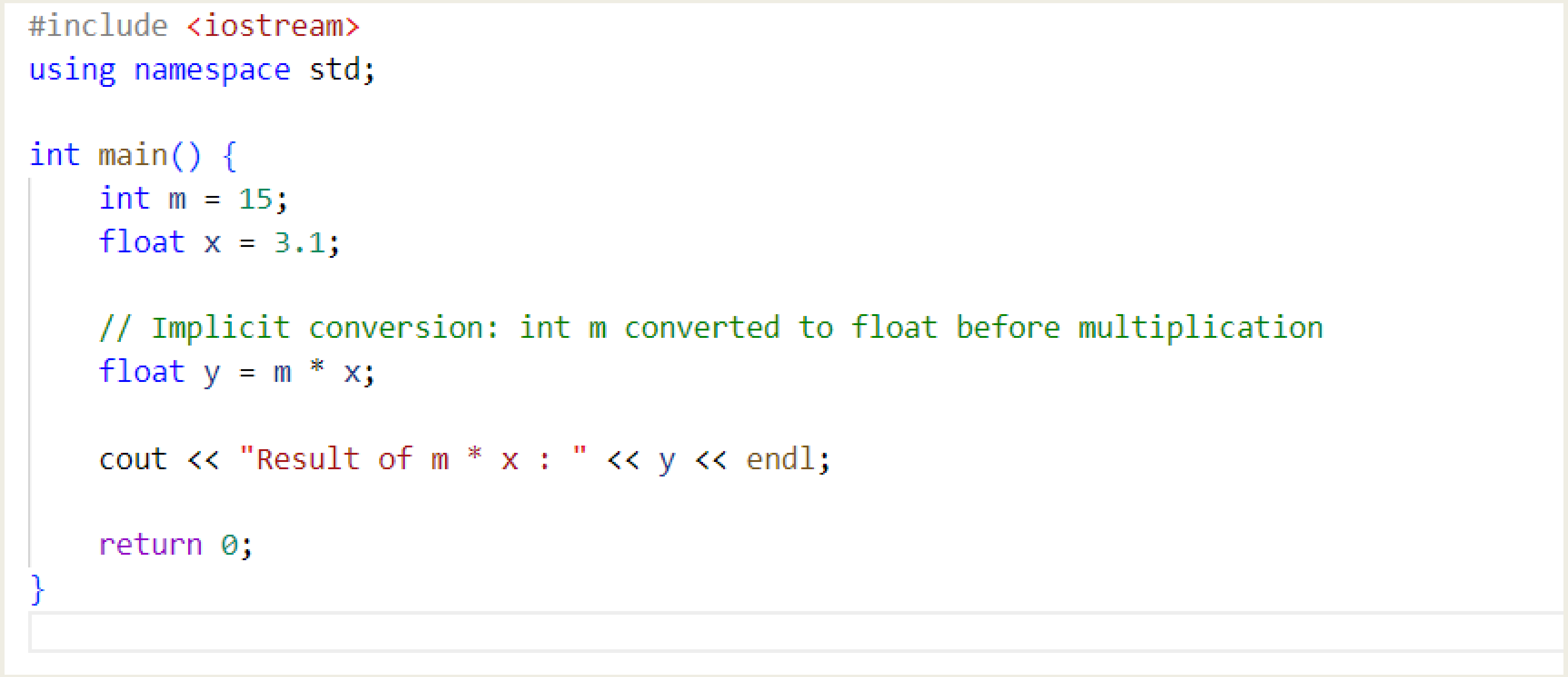
1. Implicit Conversion (Automatic Conversion):
2. Explicit Conversion (Type Casting):

## Implicit conversion

- This happens **automatically** by the compiler when you mix different data types.
- The compiler converts a value from one type to another **without you having to do anything.**
- Implicit conversion usually happens from smaller to larger or form less precise to more precise types (like int to float) to avoid losing data.
- However, implicit conversion does not happen from float to int, because converting from a floating-point type to an integer type can cause data loss (the decimal part would be lost).
- In such cases, you must use explicit type casting to tell the compiler you want to convert a float to an int.

Note : In explicit conversion also their might be the loss of the data but  the person is telling compiler so for converting from higher to lower always use explicit conversion

## Implicit conversion Example



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

int main() {
    int m = 15;
    float x = 3.1;

    // Implicit conversion: int m converted to float before multiplication
    float y = m * x;

    cout << "Result of m * x : " << y << endl;

    return 0;
}
```

## Explicit conversion

This process is also called as a type casting and it is user defined which means it is type casted by a programmer.

You manually tell the compiler to convert one data type to another

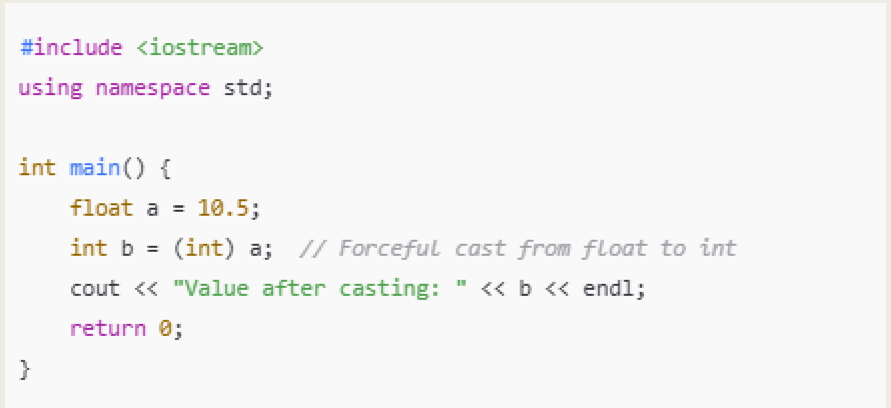
Useful when converting from a larger to smaller  or both unlike implicit conversion

In c++ explicit conversion can be done in a two ways:

1. **Converting by assignment:** This is done by explicitly (Directly) defining  the required type Infront of the expression in parenthesis . This can be also called as a forceful casting.

Syntax: (type) expression where type indicate the data type to which the final result is to be converted.

Example:



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

int main() {
    float a = 10.5;
    int b = (int) a;  // Forceful cast from float to int
    cout << "Value after casting: " << b << endl;
    return 0;
}
```

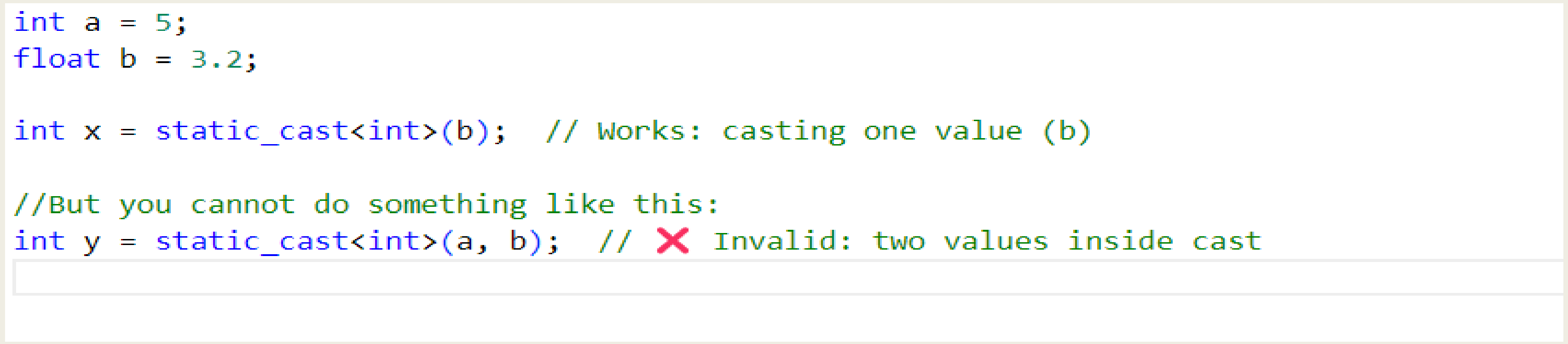
# Explicit conversion

2. Casting using cast operator:

A **cast operator** is a special operator in C++ that **converts one data type into another.**

It is a unary operator which means it works with only one value.

Cast operators help you control **when** the conversion happens—either at **compile time** or **run time**, depending on the cast you use.



```
int a = 5;
float b = 3.2;

int x = static_cast<int>(b);  // Works: casting one value (b)

//But you cannot do something like this:
int y = static_cast<int>(a, b);  // ❌ Invalid: two values inside cast
```

The most general cast operator supported by c++ compilers are as follows:

- ➢ Static cast
- ➢ Dynamic cast
- ➢ Const cast
- ➢ Reinterpret cast

Note: Each cast operator in C++ is designed for **different situations** and **different types of conversions which is not mentioned in syllabus and not needed as well**

## Manipulators (Important)

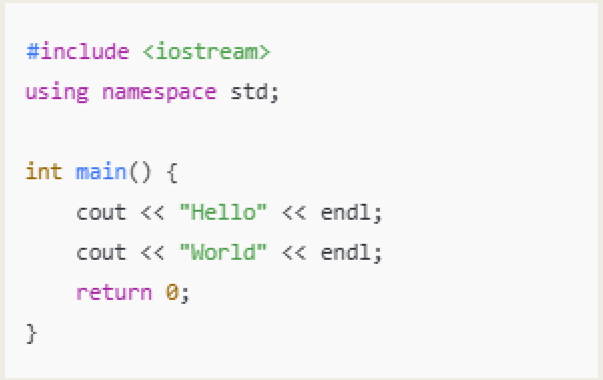
**Manipulators** are **operators** used in C++ to **format the display of data** in the output stream. They help control **how the output appears**

Manipulators are operators used in C++ (mostly with cout) to control the formatting of output.

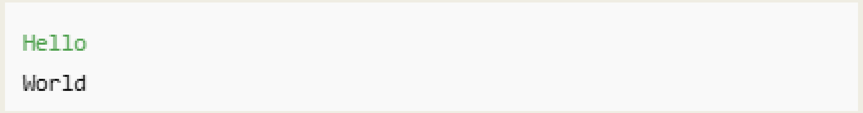
The most commonly used manipulator in c++ are endl and setw.

### endl:

The endl manipulator works like the \n (newline character) in C. When used in an output statement, endl inserts a linefeed (newline) into the output



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

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



```
Hello
World
```

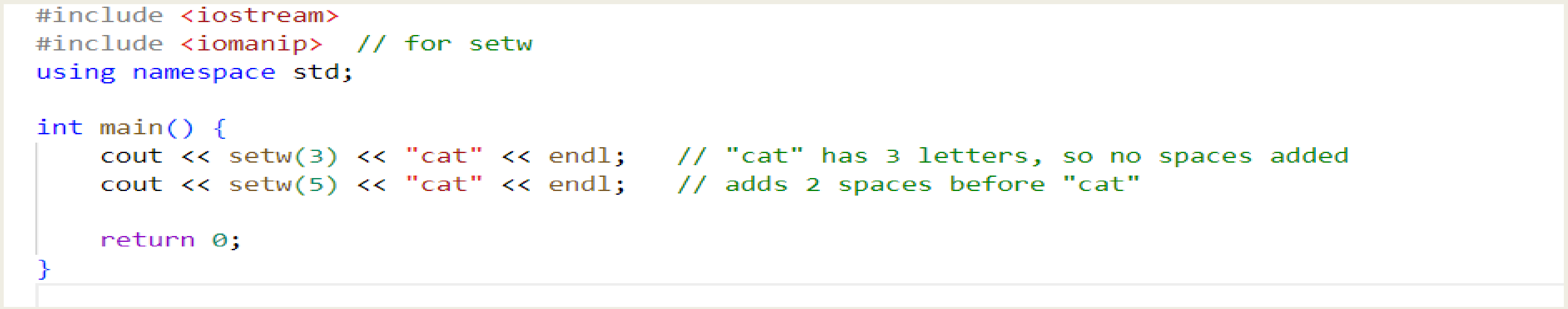
## Manipulators (Important)

Setw:

The setw manipulator helps to shift the output to the right by setting the width of the next output. It adds spaces on the left to move the text to the right as needed.

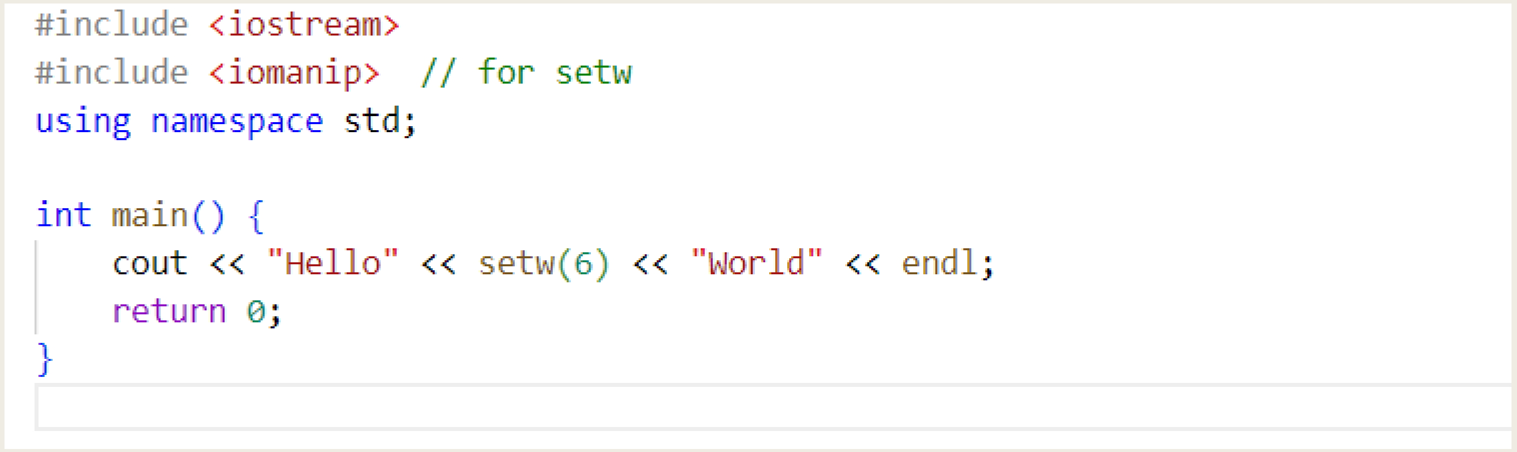
The amount of shift setw does depends on the length of the string or number you print next.

- *If the width set by setw is larger than the string's length, spaces are added on the left.*
- *If the width is equal or smaller, no extra spaces are added.*
- *To use setw we need to include a header file <iomanip> but endl is a part of <iostream> header file*

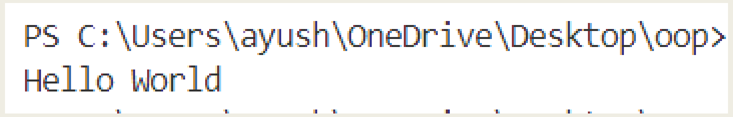


```
#include <iostream>
#include <iomanip>  // for setw
using namespace std;

int main() {
    cout << setw(3) << "cat" << endl;   // "cat" has 3 letters, so no spaces added
    cout << setw(5) << "cat" << endl;   // adds 2 spaces before "cat"

    return 0;
}
```

## Another example of setw

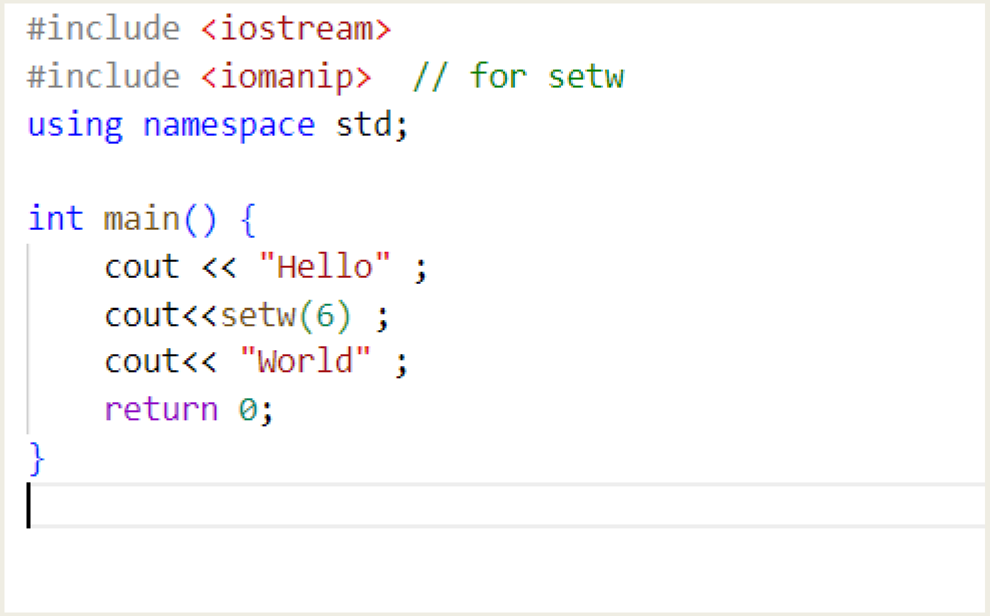


```cpp
#include <iostream>
#include <iomanip>  // for setw
using namespace std;

int main() {
    cout << "Hello" << setw(6) << "World" << endl;
    return 0;
}
```

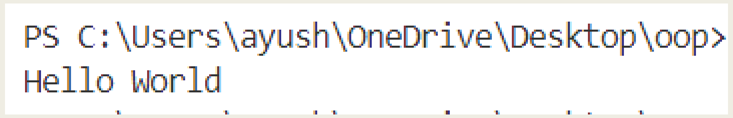


```
PS C:\Users\ayush\OneDrive\Desktop\oop>
Hello World
```

## Another example of setw



```
#include <iostream>
#include <iomanip>  // for setw
using namespace std;

int main() {
    cout << "Hello" ;
    cout<<setw(6) ;
    cout<< "World" ;
    return 0;
}
```



```
PS C:\Users\ayush\OneDrive\Desktop\oop>
Hello World
```

## Dynamic memory allocation with new and delete keywords

Dynamic memory allocation means reserving memory at runtime  instead of compile time.

It is particularly useful when we don’t know how much memory to allocate during the runtime and to reduce the wastage of memory the memory is generally allocated at a run time

When you allocate memory for variables normally (like int a = 5;), it is generally stored in the stack region.

However, if you allocate memory dynamically at it is allocated in the heap region. Heap section in a memory is designed for a flexible memory allocation and can increase and decrease the memory requirement as per the need

In c++ the memory is allocated using new keyword and deallocated using a delete keyword

## Dynamic memory allocation with new and delete keywords

New operator:

- ➢ The new operator is used for dynamic memory allocation in the heap section of memory.
- ➢ It allocates memory at runtime and returns the address of the allocated memory to a pointer variable.
- ➢ If sufficient memory is available, the new operator initializes the memory and returns its address.
- ➢ If there's not enough memory, the new operator may throw an exception.
- ➢ We can also allocate memory for array dynamically using the new operator

Delete operator:

- ■ When memory is allocated dynamically using the new operator, it is the programmer's responsibility to free that memory when it is no longer needed.
- ■ C++ provides the delete operator to deallocate memory allocated by new.
- ■ Use delete for  deallocation of memory for single variables.
- ■ Use delete[] for deallocation of memory for arrays.

## Dynamic memory allocation with new and delete keywords

Syntax of allocation of memory

For a single variable

```
data_type*  pointer_name = new data_type;

int* p = new int;     // Allocates memory for one integer i.e. 4 byte
```

For an array:

```
data_type* pointer_name = new data_type[size];

int* arr = new int[5];  // Allocates memory for 5 integers i.e. 5*4= 20 byte
```

## Dynamic memory allocation with new and delete keywords

Syntax of deallocation of memory

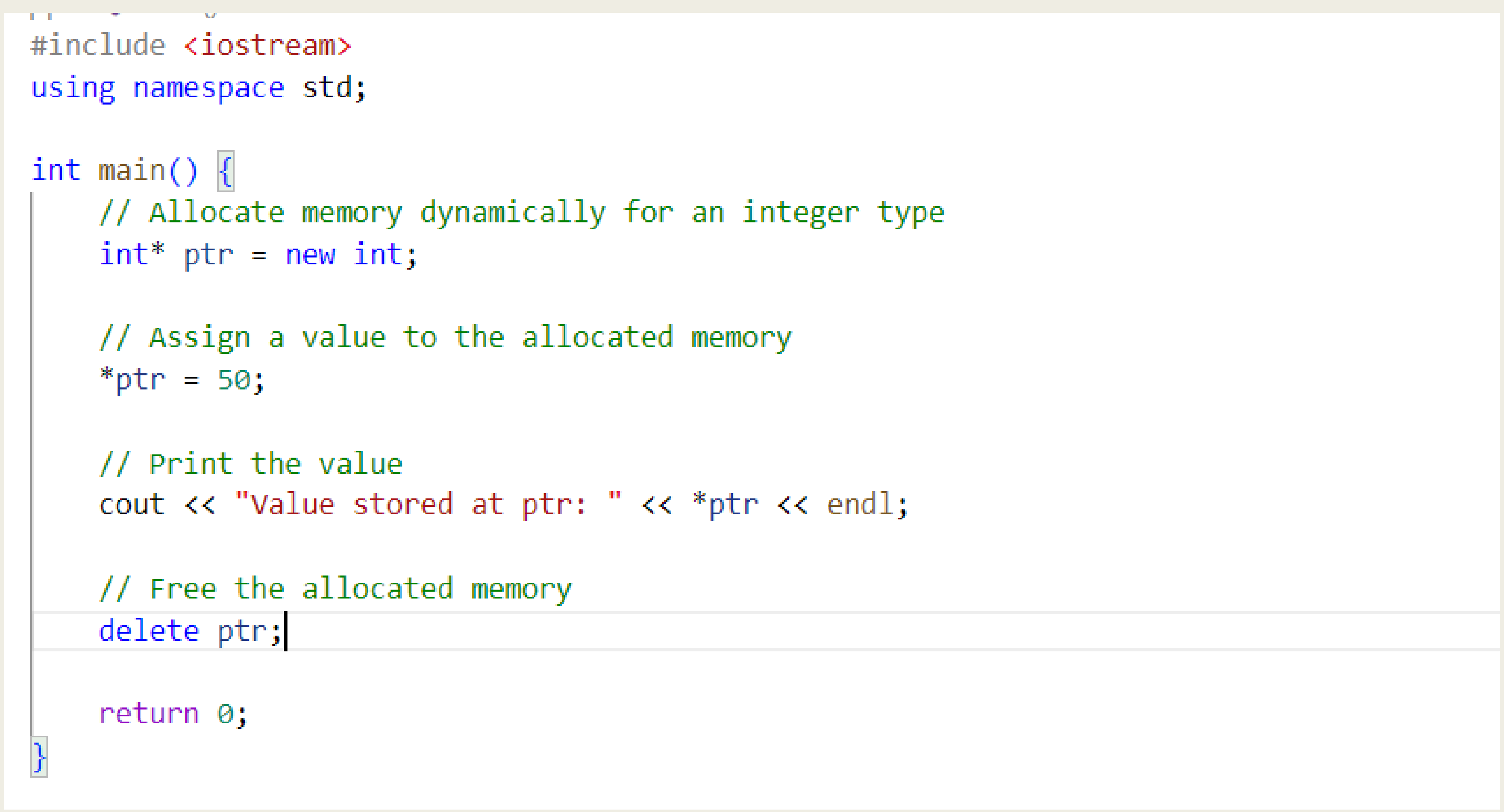
For a single variable

```
delete pointer_name;
```

For an array:

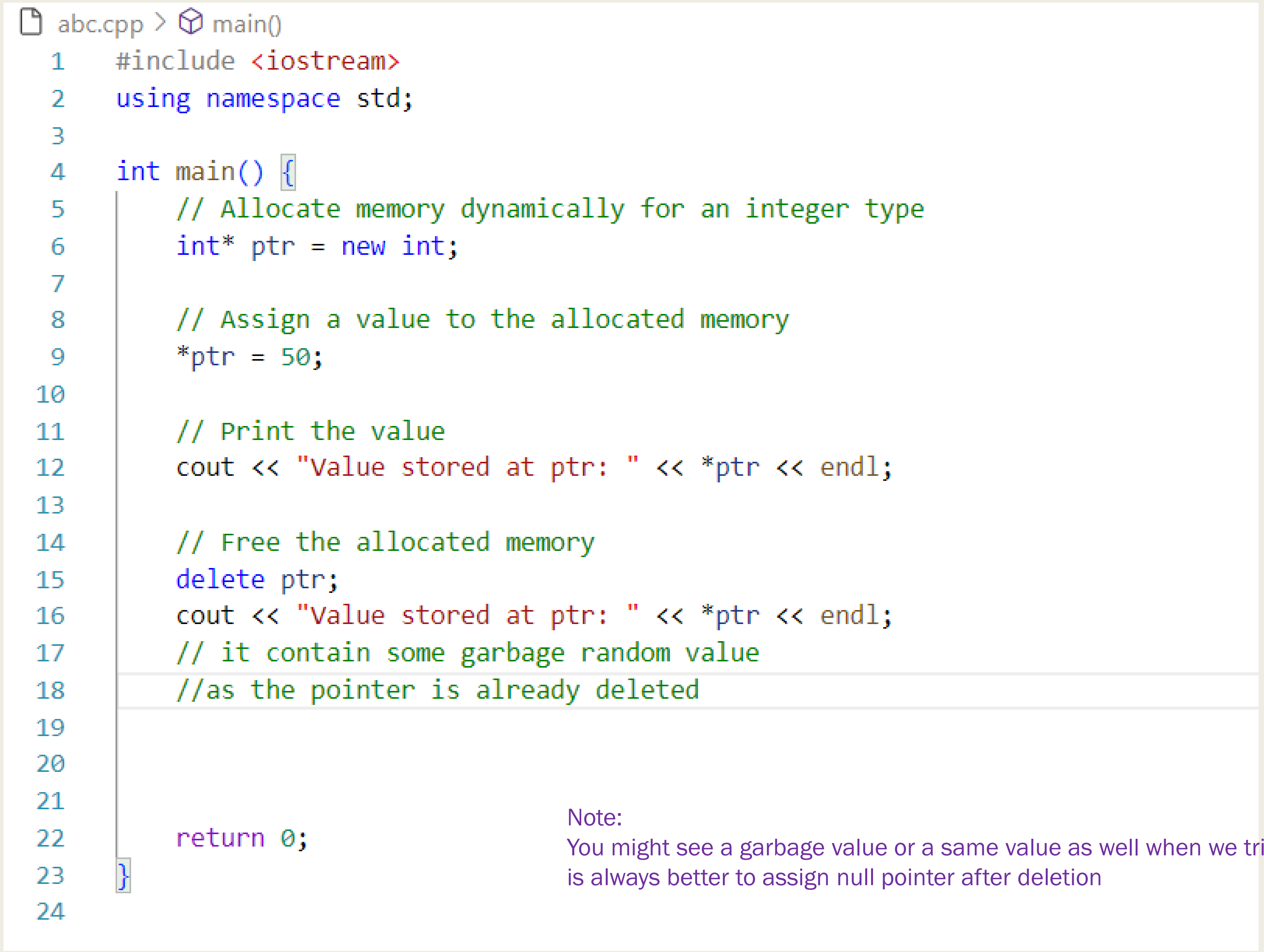
```
delete[] pointer_name;
```

## Example of memory allocation (single variable)



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

int main() {
    // Allocate memory dynamically for an integer type
    int* ptr = new int;

    // Assign a value to the allocated memory
    *ptr = 50;

    // Print the value
    cout << "Value stored at ptr: " << *ptr << endl;

    // Free the allocated memory
    delete ptr;

    return 0;
}
```

# Example of memory allocation (Try this)

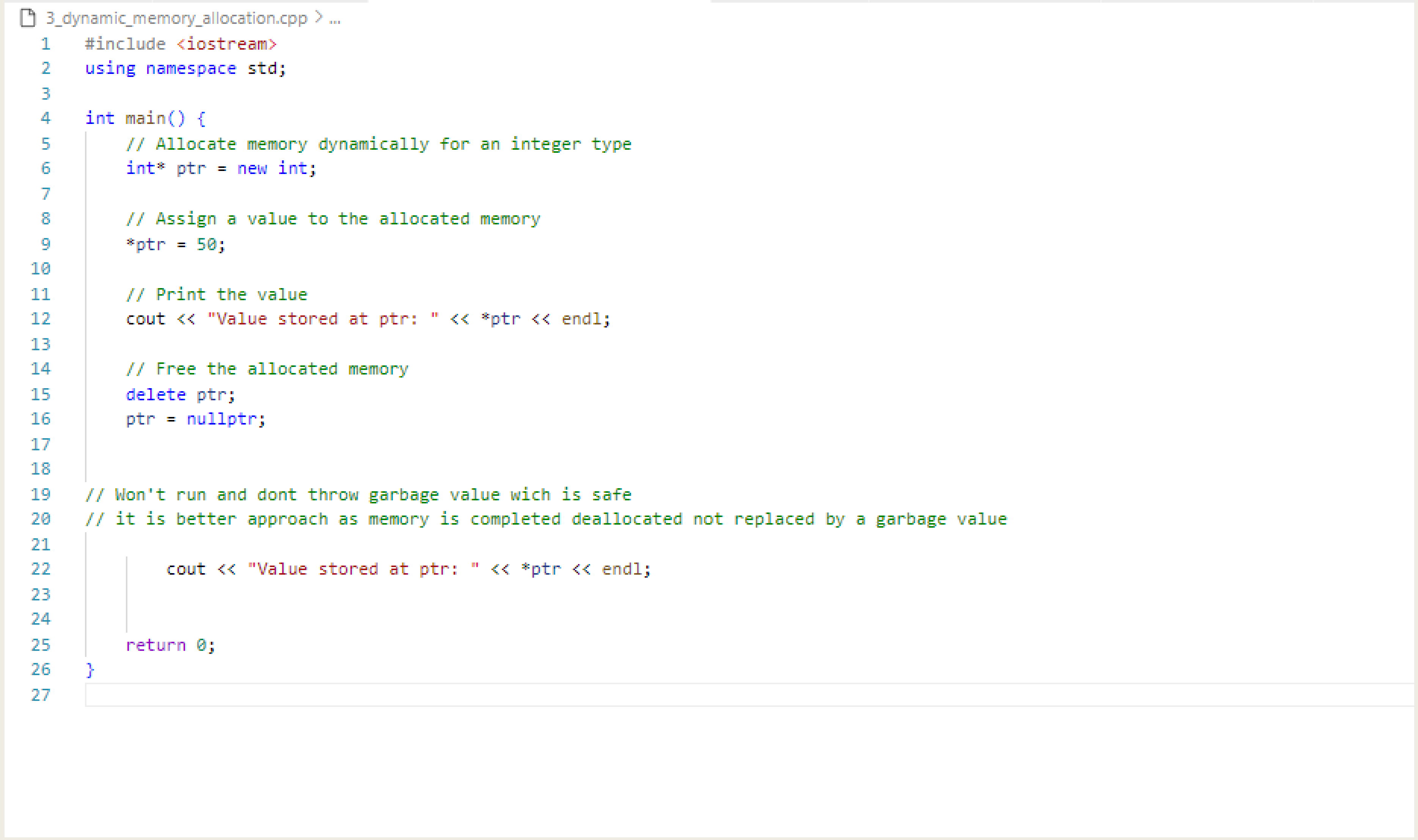


abc.cpp > main()

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

int main() {
    // Allocate memory dynamically for an integer type
    int* ptr = new int;

    // Assign a value to the allocated memory
    *ptr = 50;

    // Print the value
    cout << "Value stored at ptr: " << *ptr << endl;

    // Free the allocated memory
    delete ptr;
    cout << "Value stored at ptr: " << *ptr << endl;
    // it contain some garbage random value
    //as the pointer is already deleted



    return 0;
}

```

Note:
You might see a garbage value or a same value as well when we tries to access after deletion so it is always better to assign null pointer after deletion

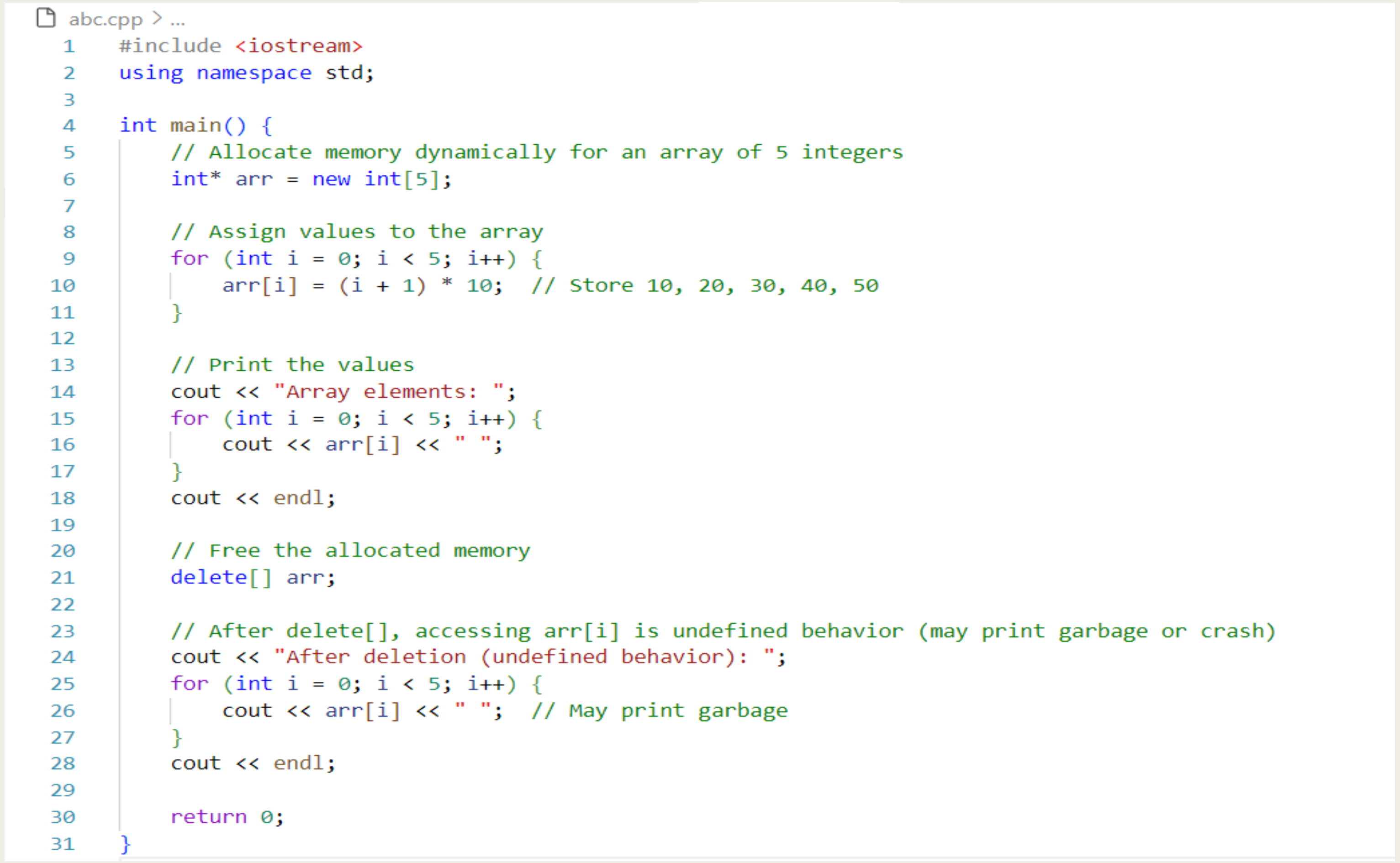
## Example of memory allocation  and deallocation using null pointer



3_dynamic_memory_allocation.cpp > ...

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

int main() {
    // Allocate memory dynamically for an integer type
    int* ptr = new int;

    // Assign a value to the allocated memory
    *ptr = 50;

    // Print the value
    cout << "Value stored at ptr: " << *ptr << endl;

    // Free the allocated memory
    delete ptr;
    ptr = nullptr;


// Won't run and dont throw garbage value wich is safe
// it is better approach as memory is completed deallocated not replaced by a garbage value

        cout << "Value stored at ptr: " << *ptr << endl;


    return 0;
}

```

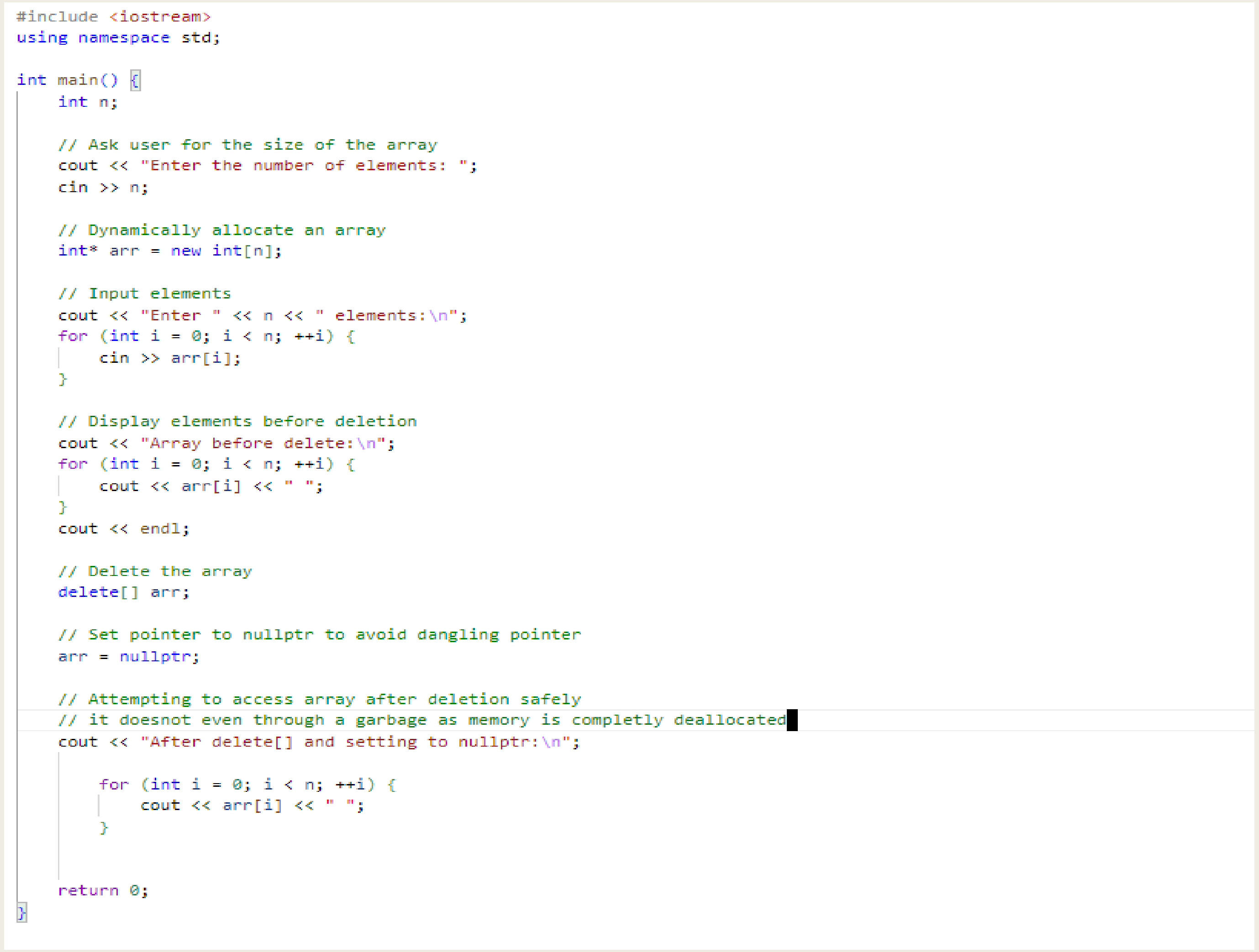
## Memory allocation and deallocation for array



abc.cpp > ...

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

int main() {
    // Allocate memory dynamically for an array of 5 integers
    int* arr = new int[5];

    // Assign values to the array
    for (int i = 0; i < 5; i++) {
        arr[i] = (i + 1) * 10;  // Store 10, 20, 30, 40, 50
    }

    // Print the values
    cout << "Array elements: ";
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Free the allocated memory
    delete[] arr;

    // After delete[], accessing arr[i] is undefined behavior (may print garbage or crash)
    cout << "After deletion (undefined behavior): ";
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";  // May print garbage
    }
    cout << endl;

    return 0;
}
```

## Memory allocation and deallocation for array using null pointer after deletion



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

int main() {
    int n;

    // Ask user for the size of the array
    cout << "Enter the number of elements: ";
    cin >> n;

    // Dynamically allocate an array
    int* arr = new int[n];

    // Input elements
    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    // Display elements before deletion
    cout << "Array before delete:\n";
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Delete the array
    delete[] arr;

    // Set pointer to nullptr to avoid dangling pointer
    arr = nullptr;

    // Attempting to access array after deletion safely
    // it doesnot even through a garbage as memory is completly deallocated
    cout << "After delete[] and setting to nullptr:\n";

        for (int i = 0; i < n; ++i) {
            cout << arr[i] << " ";
        }


    return 0;
}
```

Note:
Assigning the null pointer **prevents garbage access** and **protects your program** from undefined behavior

## Control statement (Not important)

- A **control statement** is a statement in a program that **alters the normal flow of execution.** It decides **when and how** specific parts of the program are executed, repeated, or skipped.
- A program usually runs **line by line**, from **top to bottom Like line 1, line 2 , line 3 and line 4**
- Control statements interrupt this linear flow and make the program:
  - *Jump to a different line (like goto or return)*
  - *Repeat some lines (like loops: for, while)*
  - *Skip some lines (like if-else or continue)*
- In some program we may want to execute only a part of program or we may want to execute the same statement several times, this is possible with the help of different control statements.
- Control statement define how the control is transferred to other part of program.

## Control statement (Not important)

As in c , in c++ also mainly statement and looping statement as follows

**Decision making statements:**

Simple if statement

If… else statement

Nested if statement

Switch case statement

Looping statements:

While loop

Do while loop

For loop

Nested for loop

Note: try yourself If needed as you have already gain the full knowledge of this in c.

## Function overloading (Important)

**Function overloading** is a feature in programming where **multiple functions** can have the **same name** but **different parameters** (number, type, or order).
This allows you to **use the same function name** for different purposes.

In c++ also we can use the concept of function overloading.

The function having different number or type of parameter are known as overloading of function.

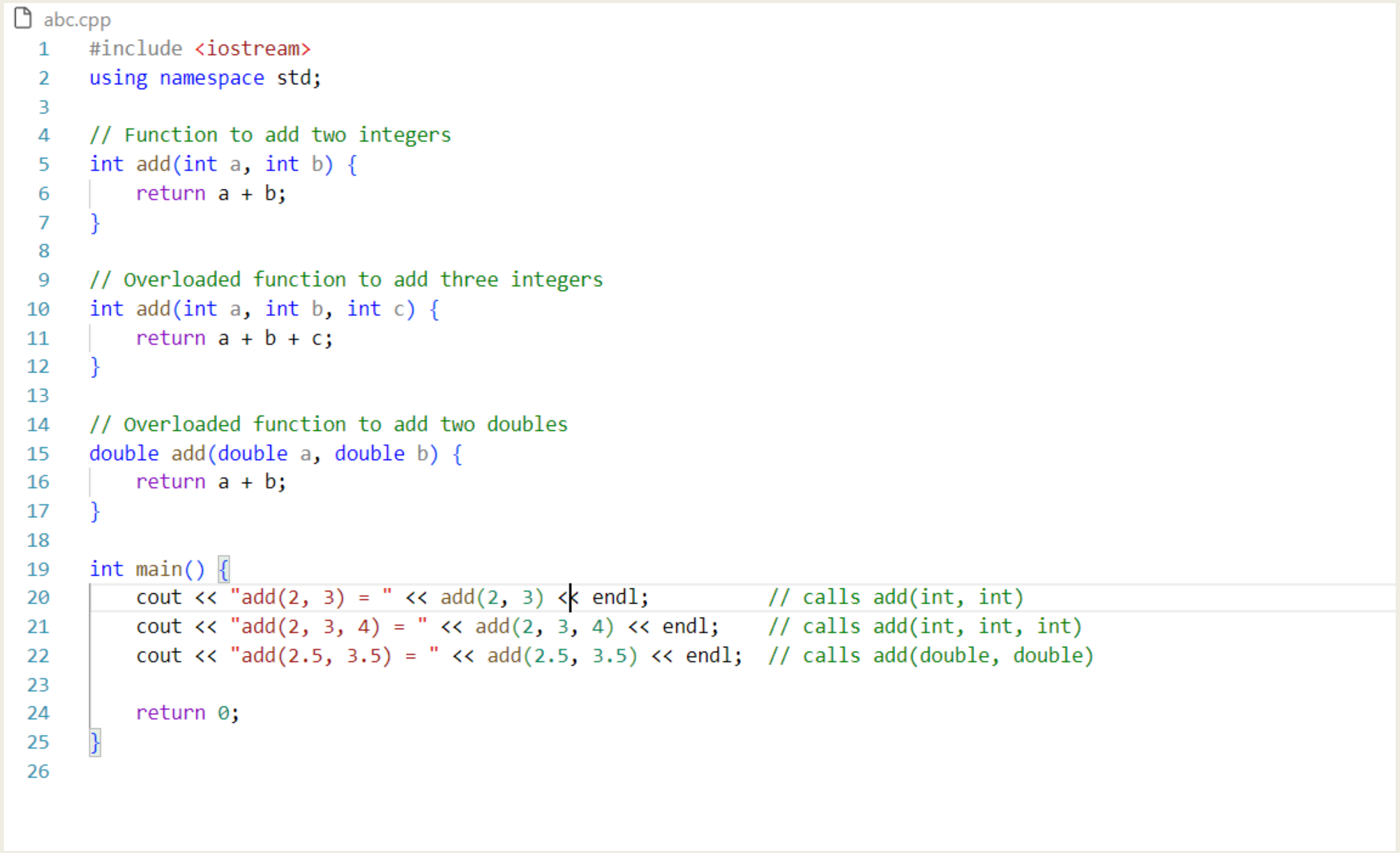
Int test(){  }

Int test (int a){}

Here two function are overloaded function because argument passed to these function are different

Notice: The return type of different function can be same or may be different by the parament must be different

## Function overloading (Important)



abc.cpp

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

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Overloaded function to add three integers
int add(int a, int b, int c) {
    return a + b + c;
}

// Overloaded function to add two doubles
double add(double a, double b) {
    return a + b;
}

int main() {
    cout << "add(2, 3) = " << add(2, 3) << endl;          // calls add(int, int)
    cout << "add(2, 3, 4) = " << add(2, 3, 4) << endl;    // calls add(int, int, int)
    cout << "add(2.5, 3.5) = " << add(2.5, 3.5) << endl;  // calls add(double, double)

    return 0;
}

```

## Inline functions (Important)

- Inline function are a c++ enhancement feature to increase the execution of time of a program.
- An **inline function** is a function where the compiler tries to **replace the function call with the actual function code** directly at the place where the function is called.
- Inline functions are declared using the inline keyword in C++

Note: It is only applicable for a small function. If function is big then compiler can ignore the inline request and treat the function as a normal function

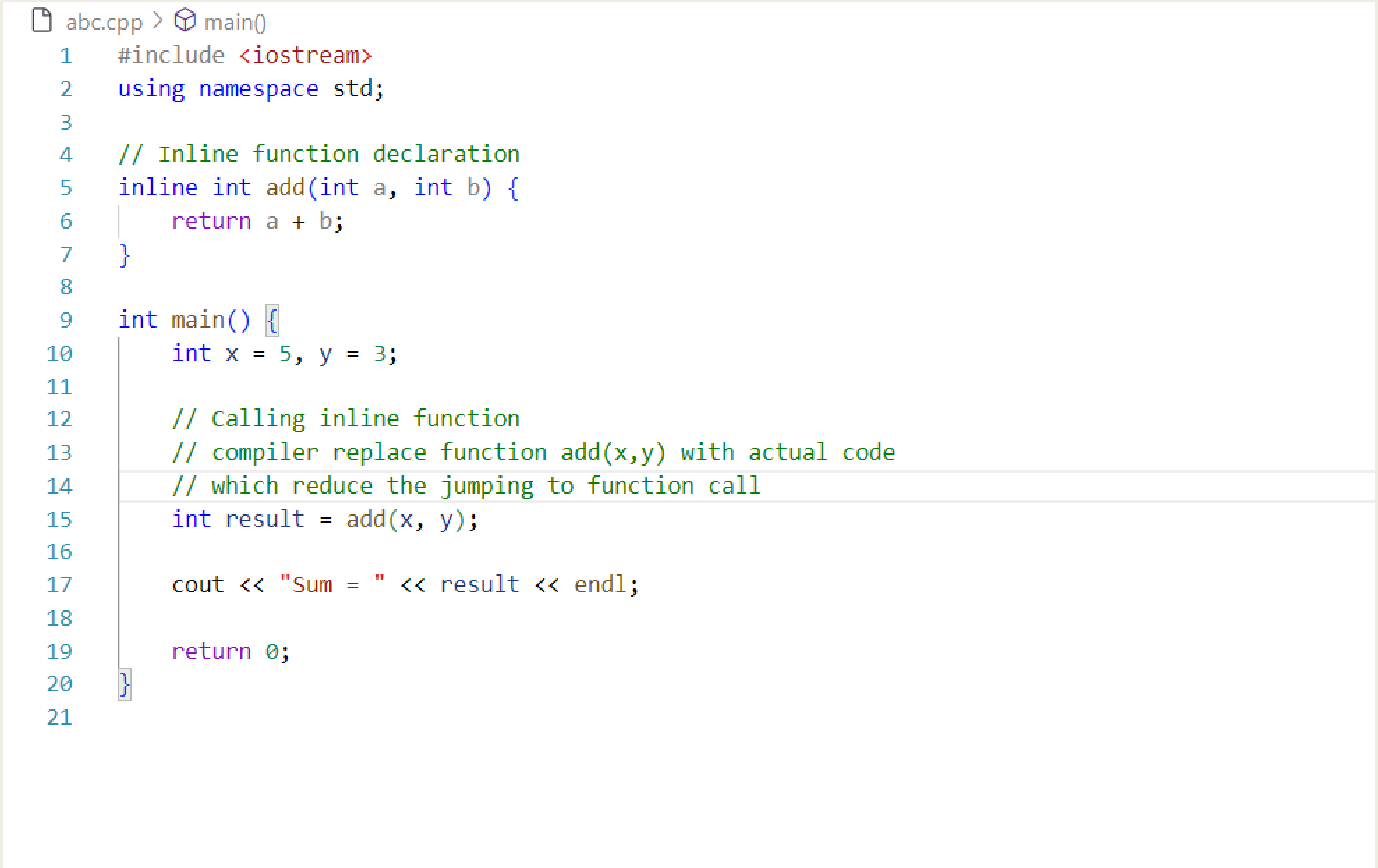
Syntax to make the function inline

```
inline return_type function_name(parameters) {
    // function body
}
```

**Simple example**

```
inline int add(int a, int b) {
    return a + b;
}
```

## Inline functions (Important)



abc.cpp > main()

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

// Inline function declaration
inline int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5, y = 3;

    // Calling inline function
    // compiler replace function add(x,y) with actual code
    // which reduce the jumping to function call
    int result = add(x, y);

    cout << "Sum = " << result << endl;

    return 0;
}

```

## Control statement (Not important)

Inline Functions vs Normal Functions

In writing:

- Inline functions look just like normal functions except you add the inline keyword before the function declaration.

In compilation:

- The key difference is how the compiler handles the function call:
- For inline functions, the compiler tries to replace the function call with the actual code of the function (inline expansion).
- For normal functions, the program calls the function by jumping to its code in memory.

Note: **inline functions reduce the overhead of function calls by inserting code directly, while normal functions involve actual calls and returns.**

## Contd…

Advantages of Inline Functions

- **No function call overhead:** The compiler replaces the function call with the actual code, so there's no time spent jumping to the function and returning.
- **Saves stack operations:** It avoids the overhead of pushing and popping variables on the stack during function calls.
- **Faster execution:** Because it skips the call and return steps, inline functions can run faster than normal functions.
- **Useful in embedded systems:** Inline functions can produce less code and improve performance, which is important for resource-constrained environments.

## Contd...

**Code Bloat**

=> Since the function's code is copied wherever it's called, the program size can become much larger, especially if the function is big or called many times.

**Compiler May Ignore inline**

=> The inline keyword is only a hint to the compiler; it might ignore it for large or complex functions.

**Not Suitable for Recursive Functions**

=> Inline expansion usually doesn't work well with recursive functions.

**Harder to Debug**

=> Because function calls are replaced with code, debugging inline functions can be more difficult.

**Longer Compile Time**

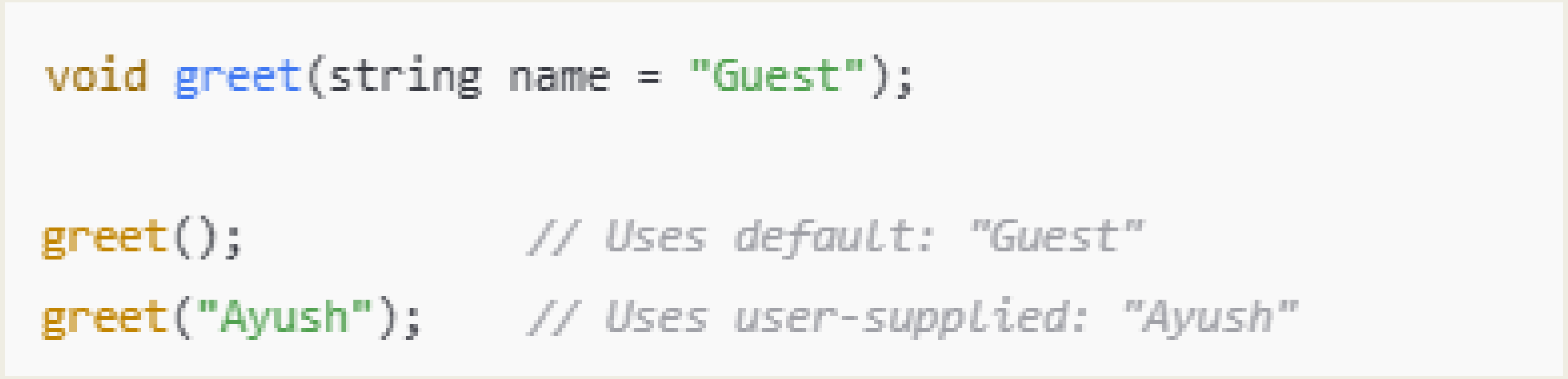
=>More code duplication means the compiler takes longer to compile the program.

**Not Always Good for Embedded Systems**

=> Some embedded systems often prioritize code size over speed, so inline functions might not be helpful there.

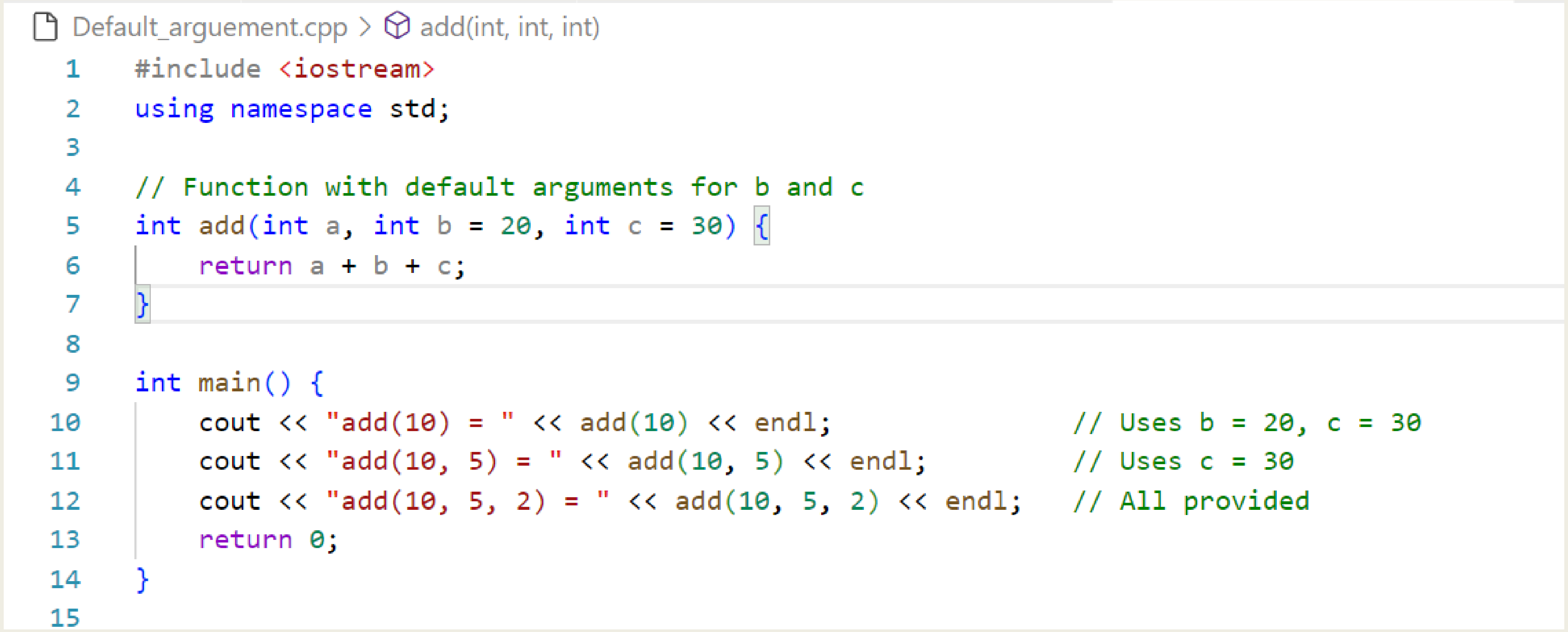
## Default Arguments

- A **default argument** is a function parameter that has a default value provided in the function declaration.
- If the caller does not supply a value for this parameter, the **default value is automatically used** by the compiler.
- However, if the caller **does supply a value**, then the **user-supplied value overrides** the default.



```
void greet(string name = "Guest");


greet();            // Uses default: "Guest"
greet("Ayush");     // Uses user-supplied: "Ayush"
```

## Default Arguments


Default_arguement.cpp > add(int, int, int)

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

// Function with default arguments for b and c
int add(int a, int b = 20, int c = 30) {
    return a + b + c;
}

int main() {
    cout << "add(10) = " << add(10) << endl;                // Uses b = 20, c = 30
    cout << "add(10, 5) = " << add(10, 5) << endl;          // Uses c = 30
    cout << "add(10, 5, 2) = " << add(10, 5, 2) << endl;   // All provided
    return 0;
}

```

## Call by reference / Pass by reference (Important)

**Call by reference** is a method of passing arguments to a function where the function receives the **address** (or reference) of the actual variable, not a copy of its value. This allows the function to **directly modify** the original variable in the calling program.

Sometimes, we want to change the value of a variable in the calling program by using a function. However, this is **not possible** when the **call by value** method is used. This happens because the function only works with a **copy** of the variable's value, not the actual variable itself.

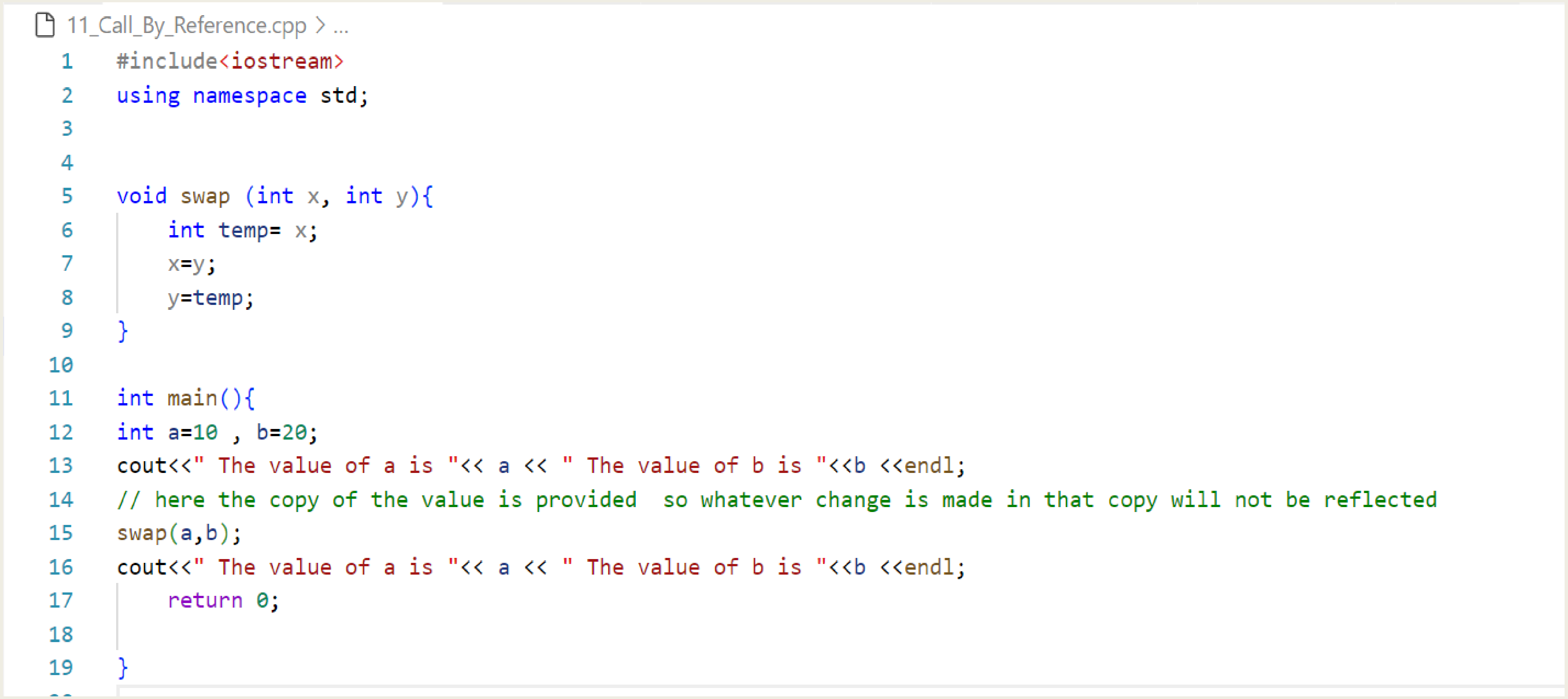
In call by value, the function does **not** have access to the original variable in the calling program — it only works on the copy of the value passed to it. Therefore, any changes made to the parameter inside the function do **not** affect the original variable.

The call by value method is fine if the function only needs to use the value but does **not** need to change the original variable in the calling program.

In summary, **call by reference** allows a function to modify the actual variables in the calling program by working with their addresses. This makes it possible to change the original data, unlike **call by value**, which only works with copies and cannot alter the original variables.

## Call By reference / Pass by reference  (Important)

Will this work??

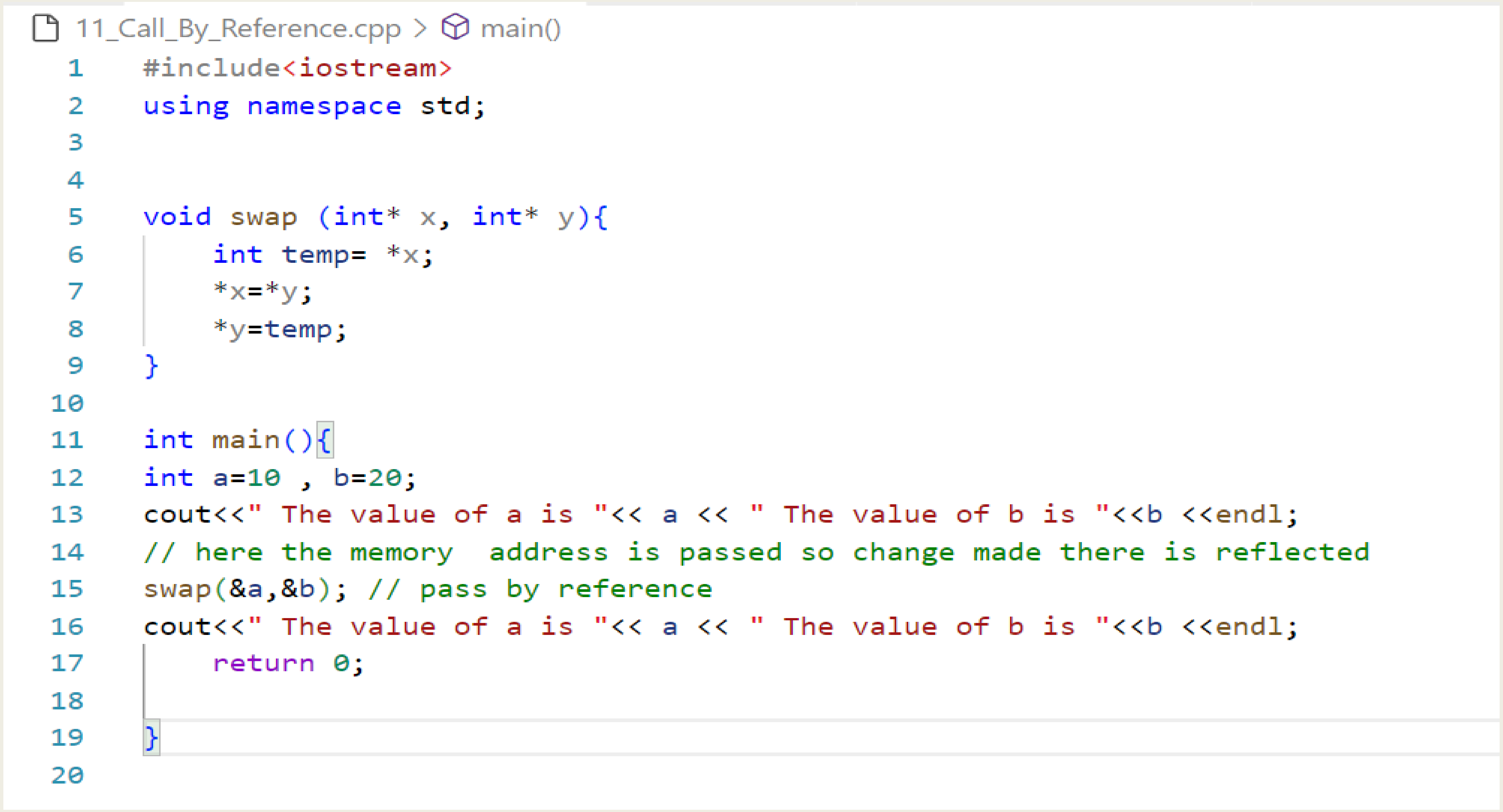


11_Call_By_Reference.cpp > ...

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


void swap (int x, int y){
    int temp= x;
    x=y;
    y=temp;
}

int main(){
int a=10 , b=20;
cout<<" The value of a is "<< a << " The value of b is "<<b <<endl;
// here the copy of the value is provided  so whatever change is made in that copy will not be reflected
swap(a,b);
cout<<" The value of a is "<< a << " The value of b is "<<b <<endl;
    return 0;

}
```

## Call By reference / Pass by reference  (Important)

Example of pass by reference


11_Call_By_Reference.cpp > main()

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


void swap (int* x, int* y){
    int temp= *x;
    *x=*y;
    *y=temp;
}

int main(){
int a=10 , b=20;
cout<<" The value of a is "<< a << " The value of b is "<<b <<endl;
// here the memory  address is passed so change made there is reflected
swap(&a,&b); // pass by reference
cout<<" The value of a is "<< a << " The value of b is "<<b <<endl;
    return 0;

}

```

## Scope resolution operator ( :  : )

The scope resolution (: :) operator is a binary operator used in c++

It helps in resolving the scope of identifiers such as variable , functions , classes .

It is primarily used to:

- *Access a global version of a variable when a local variable has the same name.*
- *Define member functions outside their class definitions.*
- *Access elements inside namespaces.*

It helps the compiler **identify the correct version** of a variable or function, especially when there is **ambiguity due to similar names** in different scopes.

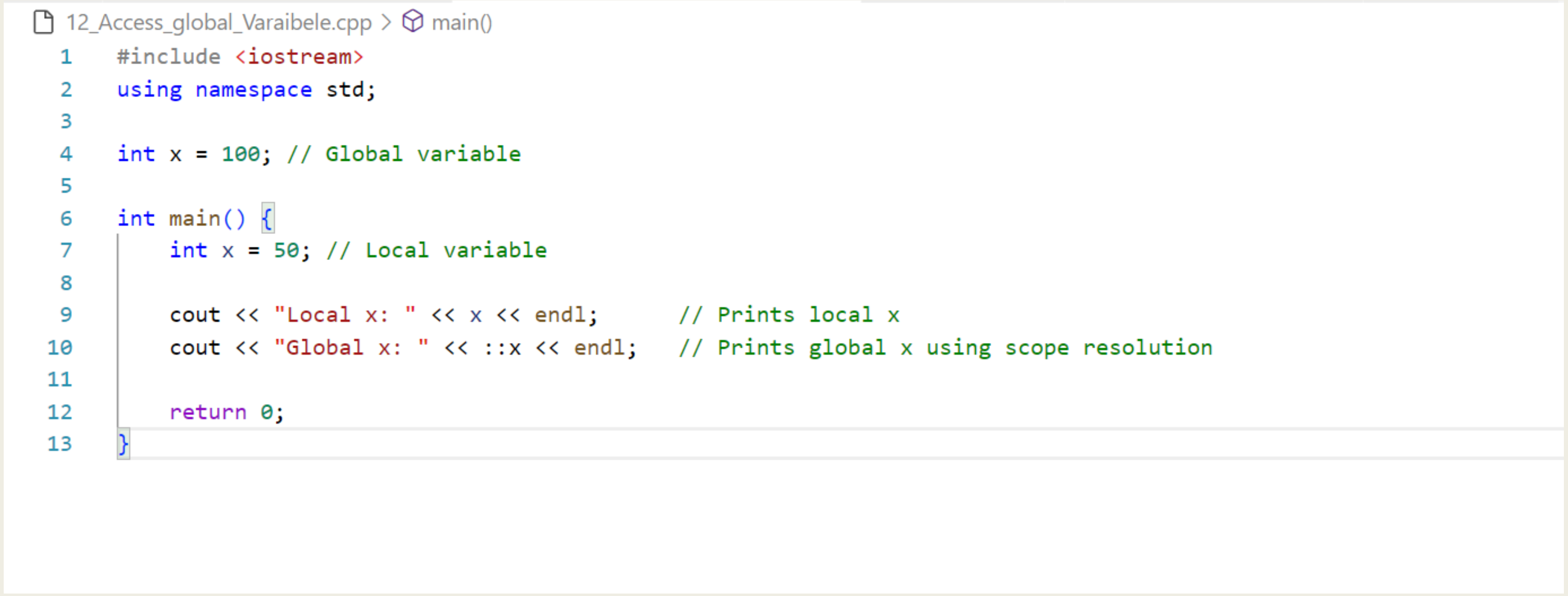
In the C programming language, if a local variable has the same name as a global variable, the global one is **hidden** and cannot be accessed directly from within the inner block.

However, C++ resolves this problem by introducing a new operator called the scope resolution operator (::). This operator allows us to access the hidden global variable even if there is a local variable with the same name.

## Scope resolution operator ( : : )

The scope resolution basically does the following two things

1. Access the global variable when we have the local variable with the same name



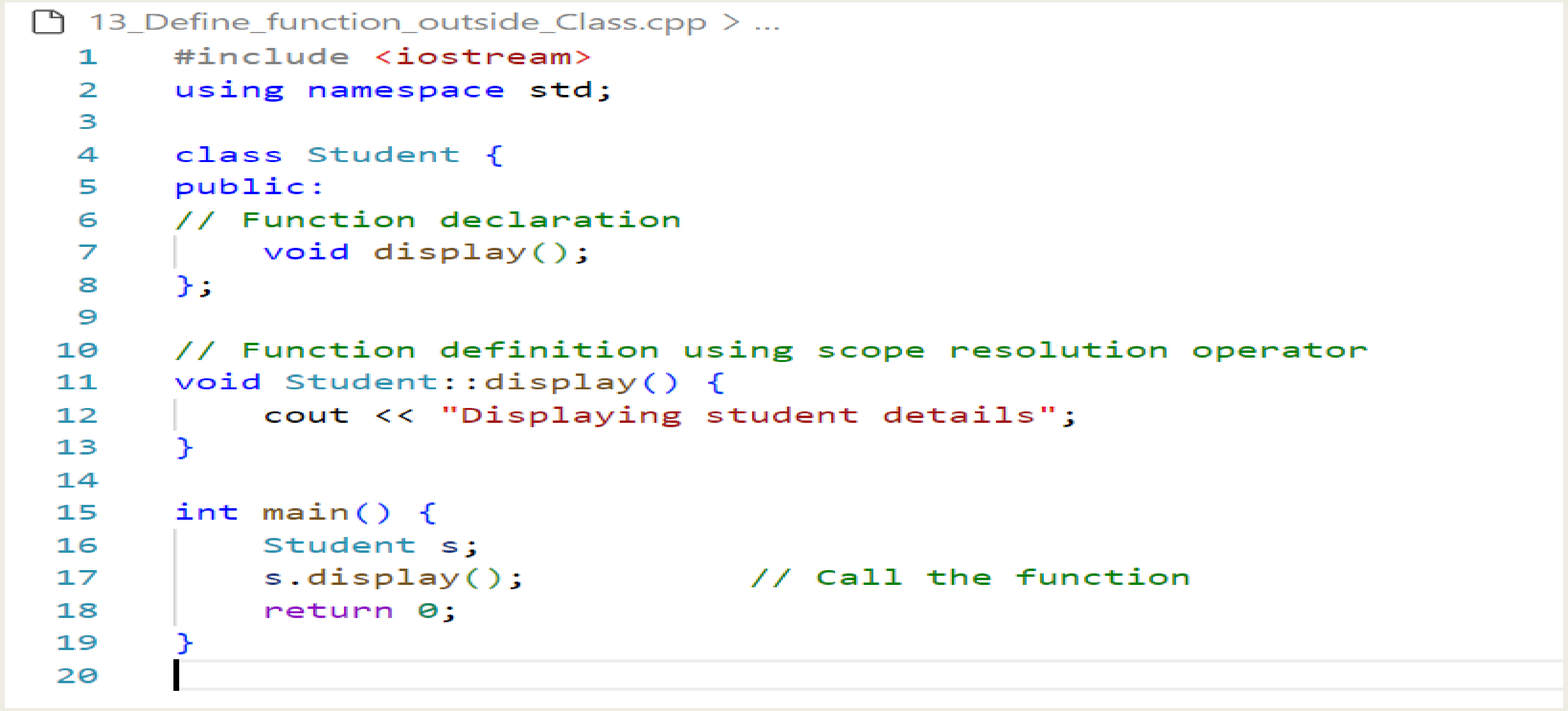
12_Access_global_Varaibele.cpp > main()

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

int x = 100; // Global variable

int main() {
    int x = 50; // Local variable

    cout << "Local x: " << x << endl;      // Prints local x
    cout << "Global x: " << ::x << endl;   // Prints global x using scope resolution

    return 0;
}
```

## Scope resolution operator ( : : )

2. Define a function outside the class.

In C++, we often **declare** member functions inside the class and **define** them outside the class to keep the class declaration clean and readable. To do this, we use the **scope resolution operator** with the class name.



13_Define_function_outside_Class.cpp > ...

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

class Student {
public:
// Function declaration
    void display();
};

// Function definition using scope resolution operator
void Student::display() {
    cout << "Displaying student details";
}

int main() {
    Student s;
    s.display();            // Call the function
    return 0;
}
```

## Type compatibility (Not important)

**Type compatibility** refers to whether or not two data types can be used together safely in operations such as assignments, function calls, or expressions.

It determines whether a value of one type can be assigned or passed to a variable or function expecting another type.

C++ enforces stricter type compatibility rules than C.

This helps maintain clarity, safety, and reliability in programs. It also supports advanced features like **function overloading**, where distinguishing between types is crucial.

The type of compatibility is categorized into the following three types:

1. Assignment Compatibility
2. Expression compatibility
3. Parameter compatibility

## Type compatibility

Assignment Compatibility:

**Assignment compatibility** refers to whether a value of one data type can be **assigned** to a variable of another data type without causing errors or unexpected behavior.

In C++, a value can be assigned to a variable only if the two types are **compatible.**

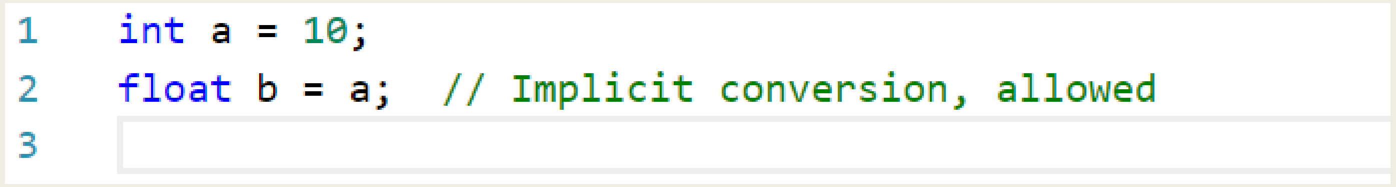
If the types are different but compatible, the compiler may perform an **implicit type conversion**

If types are incompatible or conversions might cause data loss, an **explicit type cast** is required as implicit may cause the loss of data

## Type compatibility

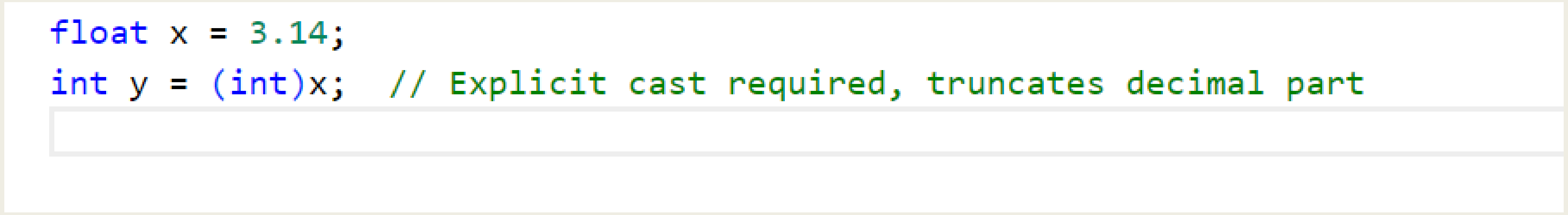
Example

int to float assignment is considered assignment compatible because the compiler can safely convert an integer to a floating-point number without losing information.



```cpp
int a = 10;
float b = a;  // Implicit conversion, allowed
```

- float to int assignment is not fully assignment compatible because converting a floating-point number to an integer may cause data loss (the decimal part is truncated). Therefore, in C++, you must use an explicit cast to show that you accept this possible loss.



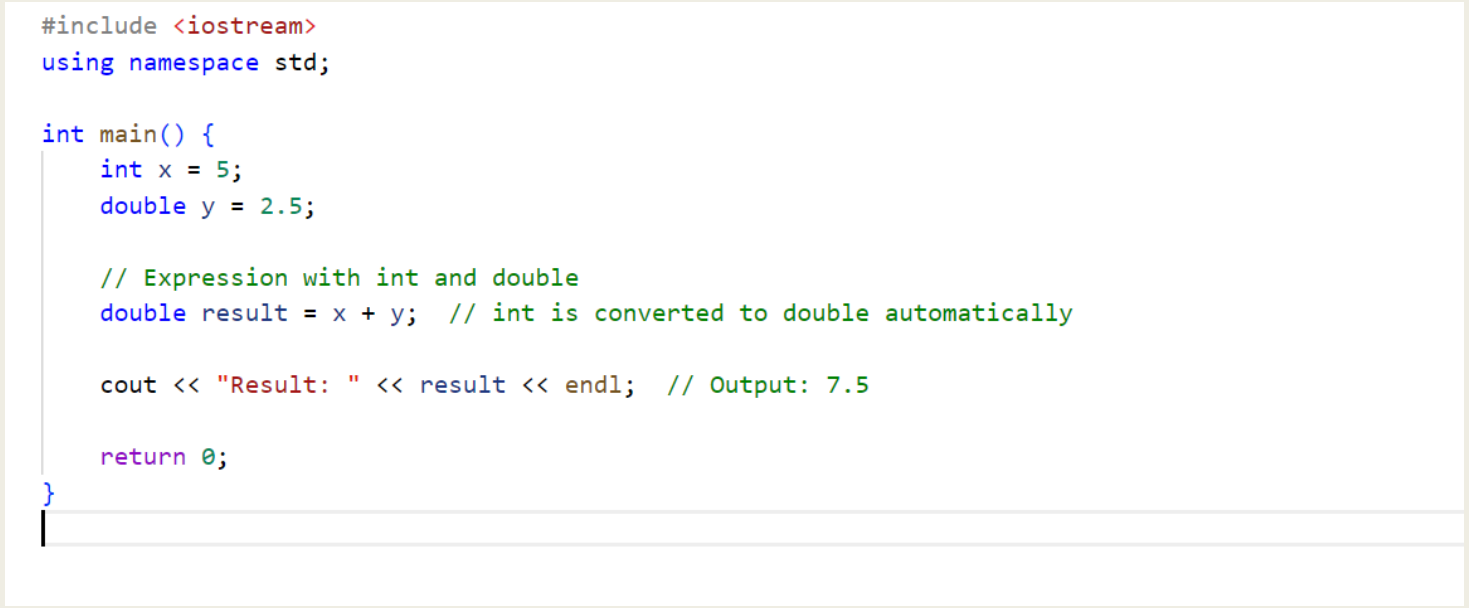
```cpp
float x = 3.14;
int y = (int)x;  // Explicit cast required, truncates decimal part
```

## Type compatibility

Expression compatibility

Expression compatibility means **whether the types involved in an expression can work together correctly** so that the operation can be performed without errors.

Example of compatibility expression

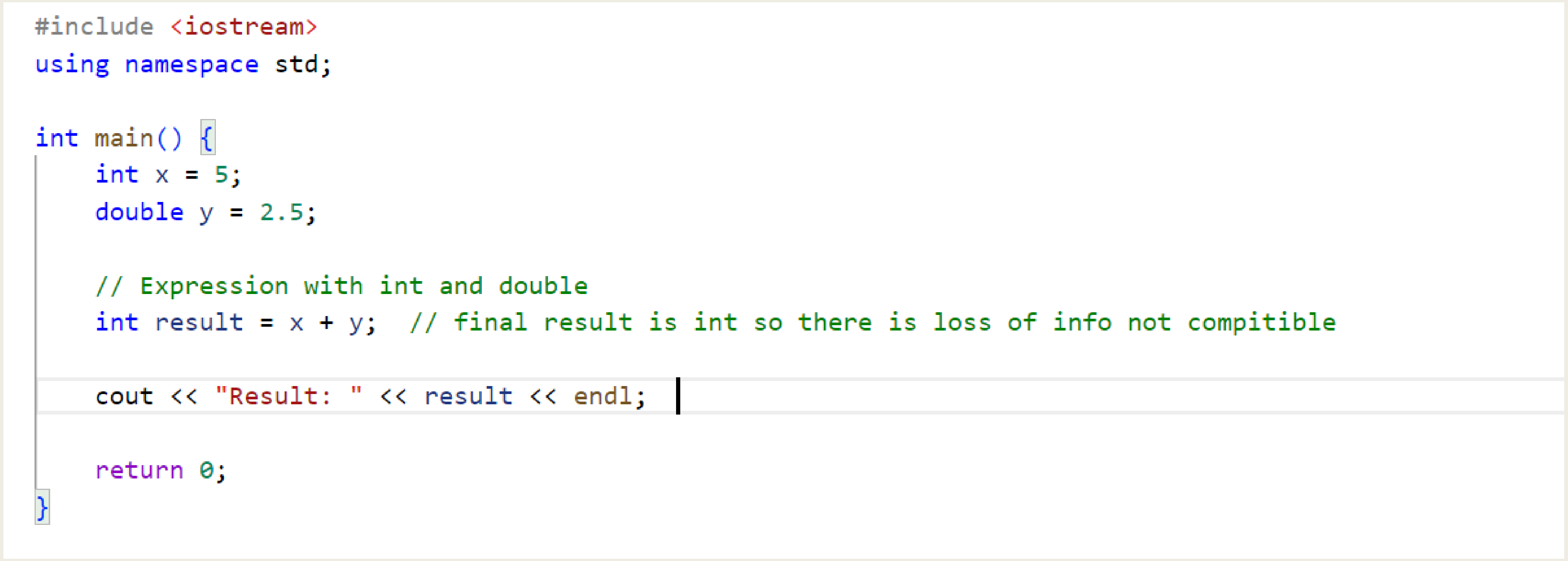


```
#include <iostream>
using namespace std;

int main() {
    int x = 5;
    double y = 2.5;

    // Expression with int and double
    double result = x + y;  // int is converted to double automatically

    cout << "Result: " << result << endl;  // Output: 7.5

    return 0;
}
```

Here during expression integer is converted into double so there is not loss of information

## Type compatibility

Example of non compatibility expression



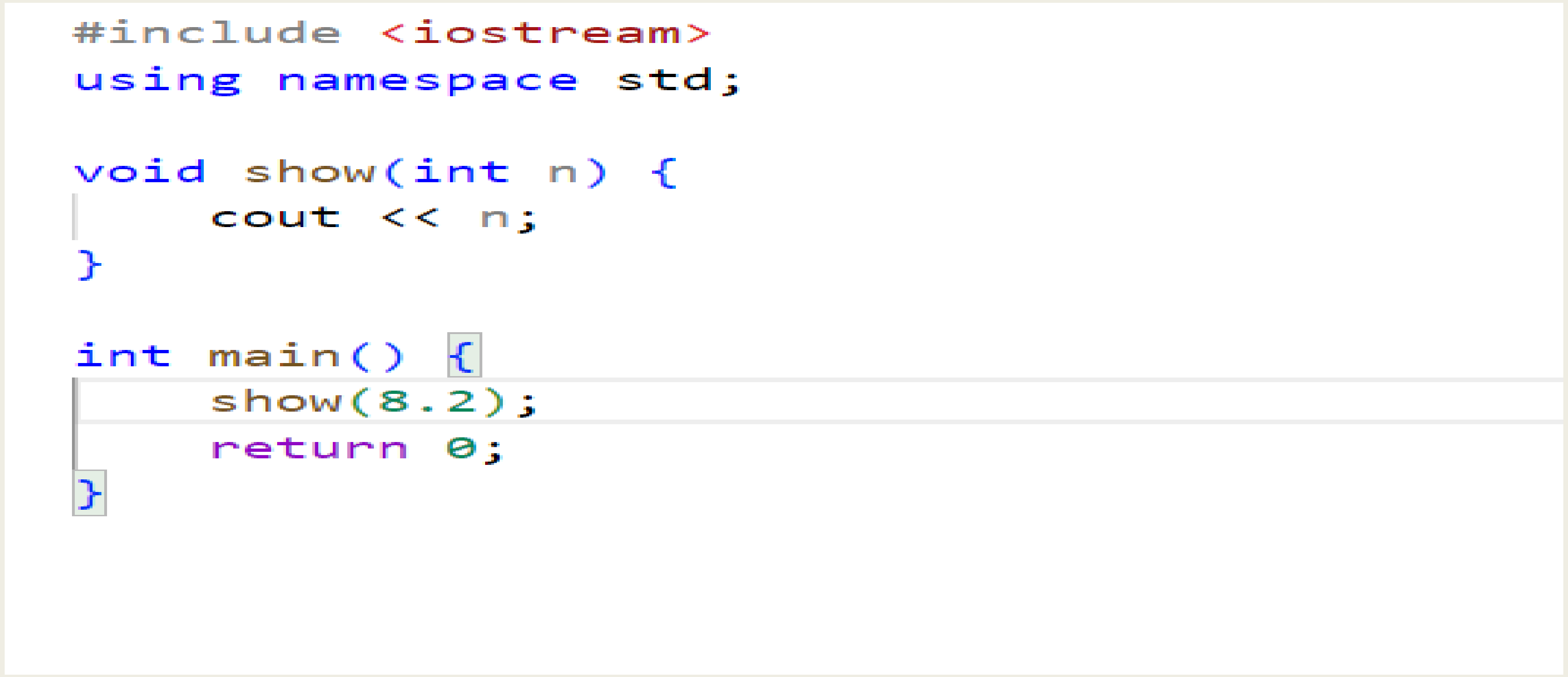
```
#include <iostream>
using namespace std;

int main() {
    int x = 5;
    double y = 2.5;

    // Expression with int and double
    int result = x + y;  // final result is int so there is loss of info not compitible

    cout << "Result: " << result << endl;

    return 0;
}
```

## Type compatibility

Parameter compatibility

It refers to whether the **arguments** passed to a function match the **parameter types** defined in the function’s declaration or definition.

If the argument type exactly matches the parameter type, it is **compatible.**



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

void show(int n) {
        cout << n;
}

int main() {
        show(8.2);
        return 0;
}
```

Here the output will be n=8 due to incompatibility in actual and formal parameter type

The End