# Introduction

## What Is C++?

C++ is a mid-level programming language that was developed as an extension of the C language. It’s known for its efficiency, versatility, and direct access to hardware, making it suitable for a wide range of applications. C++ supports both procedural and object-oriented programming paradigms, allowing developers to write code that is both efficient and organized.

Real-world applications of C++:

* Operating systems
* GUI based applications
* Distributed systems
* Database software
* Banking applications
* Advanced computations and graphics
* Embedded systems

## Advantages and Disadvantages

### Advantages

* C++ is an OOP language. That means the data is considered as **objects**. They can work with the concept of *polymorphism*, *inheritance*, *abstraction*, *encapsulation*.
* C++ program uses [multi-paradigm programming](https://www.geeksforgeeks.org/introduction-of-programming-paradigms/) (and [here](https://www.youtube.com/watch?v=B1p5OlO5tWg&ab_channel=TheCodePrism)). By **paradigm**, we mean you can program the logic, structure and procedure of program. Also, **multi-paradigm** means it follows three paradigms Generic, Imperative, Object-Oriented.
* It is a **mid-level** programming language. It can develop OS kernels, embedded systems, drivers, DB engines, graphic engines, desktop applications, games, etc.
* C++ has a **large and mature ecosystem of libraries and frameworks** that can be used to accelerate development and simplify complex tasks like network programming, graphics rendering, etc.
* C++ runs **super fast**. A program written in C++ gets **compiled directly into machine code** – without an intermediary interpretation required at runtime. By contrast, some interpreted languages – such as Java – have to rely on a Java Virtual Machine (JVM). Its code is compiled into an intermediary form, called *bytecode*, before it is converted into machine code during runtime. This intermediary interpretation consumes resources and cause speed issues.
* C++ runs on **multiple platforms** like Windows, Linux, Mac, etc. Of course, with the helps of suitable compilers.

### Disadvantages

* C++ is **not a high-level language**. Its syntax is **complex**. Hence, it can be more **difficult** to learn and to become master.
* C++ program **doesn’t support garbage collector**. It requires careful memory and resource managements. So, it’s not secure in terms of memory management. If the coder is not careful enough, the program can face issues like **memory leak**, segment fault, etc.
* C++ is a compiled language. It has to be compiled into **machine code specific to the target hardware** and OS on which it’s intended to run. By contrast, Java bytecode can run on any computer with a JVM installed, which is the basis of Java's slogan "write-once, run-anywhere".

## C vs C++

|  |  |
| --- | --- |
| **C** | **C++** |
| It is a **procedural** programming language. So, it doesn’t support classes and objects. It also doesn’t support **OOPs properties** like inheritance, polymorphism, abstraction, encapsulation. | It is a mixture of both **procedural** and **object-oriented** programming languages. So, it supports classes and objects. |
| DOES NOT supports: function overload, operator overload, namespace, STL, exception handlings … | Supports |
| **Functions CANNOT be defined inside structures**.  However, you can include *function pointers* within a structure as members of a structure. So you can achieve a form of encapsulation and associate functions with the structure.  Example:  struct MyStruct {  int data;  void (\*functionPtr)(int); // function pointer member  };    void myFunction(int value) {  printf("Value: %d\n", value);  }    int main() {  struct MyStruct myStruct;  myStruct.data = 42;  myStruct.functionPtr = myFunction;    // calling the function through the function pointer  myStruct.functionPtr(myStruct.data);    return 0;  } | Functions can be defined inside structures. |

# Command Line

## Compile and Run C/C++ Programs

**Note**

This guide is for Linux. For Windows, the principle is the same, only differences about installation and output file extensions.

### 1. Install GNU C/C++ compiler and related tools

* For Debian, Ubuntu:

$ sudo apt-get update

$ sudo apt-get install build-essential manpages-dev

**Note**: The [build-essential](http://packages.ubuntu.com/xenial/build-essential) package includes:

* gcc compiler (GNU C Compiler)
* g++ compiler (GNU C++ Compiler)
* make tool
* Other related tools for C/C++ development in Linux
* For Red Hat, CentOS, Fedora:

$ sudo yum groupinstall 'Development Tools'

### 2. Verify installation

Type the following commands to display the location and version number of the compiler:

$ whereis gcc

$ gcc --version

Sample output:



### 3. Write a "Hello World" program

* **For C**: Create a text file called helloworld.c with following content:

#include <stdio.h>

void main() {

printf("Hello, World!");

}

* **For C++:** Create a text file called helloworld.cpp with following content:

#include <iostream>

int main(void) {

std::cout << "Hello, World!";

return 0;

}

### 4. Compile the code

* **For C**: Go to your helloworld.c directory and run:

gcc helloworld.c -o helloworld

Or:

cc helloworld.c -o helloworld

Or:

# assuming that helloworld.c exists in the current directory

make helloworld

* **For C++**: Go to your helloworld.cpp directory and run:

g++ helloworld.cpp -o helloworld

Or:

# assuming that helloworld.cpp exists in the current directory

make helloworld

In both languages, if there is no error in your code, the compiler will successfully create an executable file called helloworld in the current directory, otherwise you need fix the code. To verify this, type:

$ ls -l helloworld\*

### 5. Run the executable file

Run:

$ ./helloworld

or

$ /path/to/helloworld

### Tips

#### Compile multiple source files

The syntax is as follows if the source code is in several files (such as light.c, sky.c, fireworks.c):

gcc light.c sky.c fireworks.c -o executable-file

For C++, apply the same rule as above.

# Variable

## Global vs Static

### Global

#include <stdio.h>

int g\_var = 0; // a global variable

int main()

{

printf("%d\n", g\_var);

return 0;

}

A global variable has following properties:

* It has **global scope**. So it’s **visible everywhere in the file**.

But to make it visible in the whole program (other modules and other files), you'd need to add the keyword extern, for example, extern int g\_var.

* It **lives (lifetime) until the end of the program**.
* It’s automatically **initialized to 0 or null** (depending on data types) upon program start without an explicit initialization.

### Static

A static variable has following properties:

* It can be declared **outside of all functions** or **within a function**. But in all case, it will live until the end of program.
* If a static variable is declared outside of all functions, it will have global scope.
* If it is declared within a function, it will have local scope within that function. So it cannot be called from outside the function.
* It’s **initialized to 0 or null** (depending on data types) without an explicit initialization.
* It **doesn’t work with** extern **keyword**. Because extern is used to declare a variable defined in another translation unit. But a static variable gives it internal linkage, meaning that it’s only accessible within the translation unit where it is defined.

Example:

#include <iostream>

void setNum()

{

static int num = 0;

std::cout << num << " ";

num = 3;

}

int main()

{

setNum();

setNum();

return 0;

}

Output:

0 3

## Volatile

The main purpose of volatile is to **tell the compiler that a variable can change suddenly and unexpectedly** in ways that cannot be determined at compile-time.

These changes include **external factors that are not under the control of the program**, such as:

* Hardware events
* Interrupts
* Other threads

By adding a keyword volatile before a variable declaration, you **instruct the compiler to avoid some optimizations** and generate code that **always DIRECTLY reads from/writes to the variable's MAIN MEMORY location**, ensuring that the most up-to-date value is used.

*What do the compiler optimize for variables:*

- Compilers store variables in CPU registers, so accessing to variables could be faster, compared to accessing them from main memory.

- Compilers perform various loop optimizations – loop unrolling and loop fusion – to reduce loop overhead and improve performance. More details [here](https://stackoverflow.com/a/4437555).

For example:

#include <iostream>

#include <thread>

volatile bool flag = false;

void Thread1()

{

    while (!flag)

    {

        // Do some work...

    }

    std::cout << "Thread 1: Flag is set!" << std::endl;

}

void Thread2()

{

    std::this\_thread::sleep\_for(std::chrono::seconds(2));

    flag = true;

    std::cout << "Thread 2: Set flag to true!" << std::endl;

}

int main()

{

    std::thread t1(Thread1);

    std::thread t2(Thread2);

    t1.join();

    t2.join();

    return 0;

}

Output:

Thread 2: Set flag to true!

Thread 1: Flag is set!

Without making flag volatile, the code may not work as expected. The compiler may assume that the value of flag remains constant within Thread1 loop. Thus, it may optimize the loop by caching the value of flag in a register and never update it, causing an infinite loop even when flag is set to true by Thread2.

**Notes**:

* If we remove the volatile out of the above code and run it under GCC compiler, it still works correctly. So, **whether we HAVE TO use volatile or not depends on the compiler, its settings, target platform, and the specific scenario being compiled**.
* In addition to variables, we can use volatile pointers, structures, unions, etc. The volatile structures and unions can be volatile itself, and their member variables can also be volatile.

## QAs

### Can variables be DEFINED in header file?

No, by default in C and C++. If a variable (not a class member) is DEFINED in header file and used in source file, there will be **compilation error**. Why? If that header file is included in multiple source files, it can lead to *multiple-definition* error during the linking phase. Such rule is called *One Definition Rule (ODR)*.

However, there is a way to bypass this. That is **declaring variable as extern in a header file** and define it in a source file. This is another common usecase of extern.

For example:

// File.h

extern int x; // declaration

// File.c

#include "file.h"

int x = 1; // re-declaration and definition

Of course, you have to ensure that the variable **x** is defined in ONLY ONE source file. Else, you’ll get *multiple-definition* error.

# Function

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

A function is known with various names like a **method (in class)**, a sub-routine or a procedure.

## Defining a Function

The general form of a C++ function definition is as follows:

return-type function-name(parameter-list) {

// body of the function

}

* **Return type**: A function may return a value. The *return-type* is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the *return-type* is the keyword void.
* **Function name**: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameter list**: When a function is invoked, you pass a value to the parameter. This value is referred to as an *argument*. A function can have multiple parameters or no parameter at all. The *parameter-list* refers to the type, order, and number of the parameters of a function.
* **Function body**: The function body contains a collection of statements that define what the function does.

**Example:**

Following is the source code for a function called max()*.* This function takes two parameters num1 and num2 and return the biggest of both:

// function returning the max between two numbers

int max(int num1, int num2)

{

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

## Declaring a Function

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

return\_type function\_name(parameter-list);

For the above defined function max(), following is the function declaration:

int max(int num1, int num2);

Parameter names are not important in function declaration, so following is also valid declaration:

int max(int, int);

**Function declaration is required when you define a function in one source file and you call that function in another file**. In such case, you should declare the function at the top of the file calling the function.

## Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call (or *invoke*) that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required arguments along with function name, and if function returns a value, then you can store returned value. For example:

#include <iostream>

using namespace std;

// function declaration

int max(int num1, int num2);

int main() {

// local variable declaration

int a = 100;

int b = 200;

int ret;

// calling a function to get max value

ret = max(a, b);

cout << "Max value is: " << ret << endl;

return 0;

}

// function returning the max between two numbers

int max(int num1, int num2) {

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

Output:

Max value is: 200

## Passing Arguments to Functions

Function arguments behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. While calling a function, there are two ways that arguments can be passed to a function:

|  |  |
| --- | --- |
| **Call Type** | **Description** |
| [**Call by Value**](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_value.htm) | This method copies the actual value of an argument into the parameter of the function. In this case, **changes made to the parameter inside the function have NO effect on the argument**. |
| [**Call by Pointer**](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_pointer.htm) | This method copies the address of an argument into the parameter. Inside the function, the address is used to access the actual argument used in the call. This means that **changes made to the parameter affect the argument**. |
| [**Call by Reference**](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_reference.htm) | This method copies the reference of an argument into the parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that **changes made to the parameter affect the argument**. |

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

## Default Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used. But if a value is specified, this default value is ignored and the passed value is used instead.

For example:

#include <iostream>

using namespace std;

int sum(int a, int b = 20) { // Note: You cannot write sum(int b = 20, int a)

int result;

result = a + b;

return (result);

}

int main() {

// local variable declaration:

int a = 100;

int b = 200;

int result;

// calling a function to add the values.

result = sum(a, b);

cout << "Total value is: " << result << endl;

// calling a function again as follows.

result = sum(a);

cout << "Total value is: " << result << endl;

return 0;

}

Output:

Total value is: 300

Total value is: 120

## Static Functions

Example:

static int fun(void) {

  printf("I am a static function ");

}

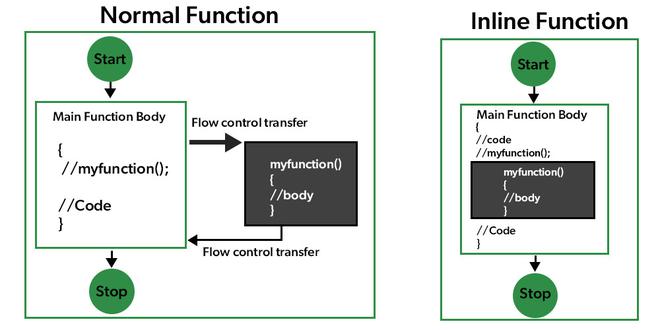
Unlike normal functions which are global, access to static functions is **restricted to the file** (translation unit) **where they are declared**.

Why use static function:

* In C and C++, making a function static **prevent it from accessing from other files**. It helps maintain modularity and reduces the risk of naming collisions with functions in other files.
* In C++, particularly in class, static member functions belong to the class itself, rather than to an instance of the class. They can be **called using the class name** (via operator ::) without the need for an object instance. For utility/common classes, it extremely useful.
* In C++, particularly in class, static member functions are allowed to **access only static data members or other static member functions**. In other words, they cannot access the non-static data members. So if you want to call any static members in a function, you’ll have to make it static.

## Inline Functions

If a function is inline, the **compiler places a copy of the code of that function at each point where the function is called at compile time**.



Using keyword inline before the function definition causes it to be an *inline function*. However, you can't redeclare a function as inline after a call to that function.

**Example:**

class Account {

    private:

        double balance;

    public:

        Account(double initial\_balance) { balance = initial\_balance; }

        double GetBalance();

        double Deposit(double Amount);

        double Withdraw(double Amount);

};

inline double Account::GetBalance() {

    return balance;

}

inline double Account::Deposit(double amount) {

    return balance += amount;

}

inline double Account::Withdraw(double amount) {

    return balance -= amount;

}

int main() {

}

**Advantages:**

* Inline functions **eliminate the function calling overheads of a traditional function** during runtime. So, they can make your program faster. What overheads? Such as pushing/popping params in the stack, jumping to the function, and returning from the function.
* The utilisation of the instruction cache increases the locality of reference.

**Disadvantages**:

* In most cases, inline function **increases binary executable program**, especially if the function is big and called many times. But in some cases, when the function size is smaller than the function call code size, they could reduce program size.

This causes other issues: more memory to load the program, more disk to store exe file, etc.

* Any change in the inline function code would **need you to recompile** the program to ensure it is updated.
* Inline functions can make it harder to trace the flow of execution through the function during debugging.

Reason: The call stack may not show the inline function as a separate frame during debugging sessions.

**Use cases**:

* Inline functions are best used for **small functions or those called many times**. Examples:
  + Simple getter and setter for class data members
  + Common/utility operations like mathematical calculations, string manipulation, type conversions
  + Overloaded operators.

Reason: Short functions are sensitive to the overhead of function calls. Longer functions spend proportionately less time in the calling and returning sequence and benefit less from inlining.

* Templated code: Inline functions are often used in conjunction with templates in C++. Templated functions are typically defined in header files, and inlining them can help avoid multiple function instantiations and linker errors.

## QAs

### Can static functions be DEFINED in header file?

**No**, by default in C and C++. If a static function (not a class member) is DEFINED in header file and used in source file, there will be **compilation error**. Why? If that header file is included in multiple source files, it can lead to *multiple-definition* error during the linking phase. Such rule is called *One Definition Rule (ODR)*.

So, you can either declare static functions in header file or source file, but must **always define them in source file**. For example:

// File.c

static void func(); // declare here

void func() {

// define here

}

### Inline functions vs. Macros?

They are similar because the function code is expanded at compile time. However, **inline functions are parsed by the compiler**, and **macros are expanded by the preprocessor**. So, there are several differences:

* Inline functions follow all rules of a normal function: same syntax, same type safety, etc.
* Expressions passed as arguments to inline functions are evaluated once. In some cases, expressions passed as arguments to macros can be evaluated more than once.

Below is two examples of a program to convert lower character to upper character. The first one is WRONG. The second one is CORRECT:

#include <stdio.h>

#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

int main() {

   char ch;

   printf\_s("Enter a character: ");

   ch = toupper( getc(stdin) );

   printf\_s( "%c", ch );

}

// Sample Input:  xyz

// Sample Output:  Z

Because of the implementation of the macro, getc is executed once to determine whether the character is greater than or equal to "a," and once to determine whether it's less than or equal to "z." If it is in that range, getc is executed again to convert the character to uppercase. It means the program waits for two or three characters when, ideally, it should wait for only one.

Now let’s move to the correct solution with inline function:

#include <stdio.h>

#include <conio.h>

inline char toupper(char a) {

   return ((a >= 'a' && a <= 'z') ? a-('a'-'A') : a );

}

int main() {

   printf\_s("Enter a character: ");

   char ch = toupper( getc(stdin) );

   printf\_s( "%c", ch );

}

// Sample Input: a

// Sample Output: A

* Unlike macros, inline is easy to debug and does not corrupt the namespaces, code as it behaves as a compiler controlled copy & paste action and not implemented forcibly.

### When the compiler reject inline functions?

**Yes.**

Inlining a function is just a suggestion to the compiler, not any mandatory command. It depends on the compiler whether to accept or reject this suggestion. The compiler is MOST LIKELY to not consider the inlining of a function under certain cases:

* When a function is **recursive**, it cannot be inlined.

Reason: Inline functions don’t maintain a piece of information on the stack which is necessary for recursion.

* When a function with a return statement is not returning any value, it cannot be inlined.

Need double check

* When a function contains a loop (for, while, or do-while), it cannot be inlined.
* W hen a function contains static variables, it cannot be inlined.
* When a function contains any switch or go-to statements.

Of course, whether be able to inline or not **heavily depends on the compiler**. So it won’t be a surprise if some modern compilers still allow inline functions in any of above cases.

# String

C++ provides two types of string representations:

* The C-style character string
* The string class type

## C-Style Character String

Originated in C and continuing to be supported in C++, C-style string uses either a **character array** or a **character pointer**.

**Library: <**[**cstring**](http://www.cplusplus.com/reference/cstring/)**> (string.h)**

### Strings as Character Arrays

They’re actually **1D arrays of characters** stored in contiguous memory locations **and** terminated by a **null** character '\0'.

The following create a string consisting of the word "Hello". To hold the null character at the end of the array, the array’s size is one more than the number of characters in "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

Or you can write the above statement as follows:

char greeting[] = "Hello";

You DON’T need to place the null character at the end of a string constant. **The C++ compiler automatically places a '\0*'* at the end of the string when it initializes the array**.

The memory layout of the above string can be visualized as follows:

+---+---+---+---+---+---+

| H | e | l | l | o | \0|

+---+---+---+---+---+---+

Following example makes use of some pre-built functions provided in cstring library:

#include <iostream>

#include <cstring>

using namespace std;

int main() {

   char str1[] = "Hello";

   char str2[] = "World";

   char str3[10];       // statically allocated sized array

   int  len ;

   // copy str1 into str3

   // char\* strcpy (char\* destination, const char\* source );

   strcpy(str3, str1);

   cout << "strcpy(str3, str1): " << str3 << endl;

   // concatenates str1 and str2

   strcat(str1, str2);

   cout << "strcat(str1, str2): " << str1 << endl;

   // total length of str1 after concatenation

   len = strlen(str1);

   cout << "strlen(str1): " << len << endl;

   return 0;

}

Output:

strcpy(str3, str1): Hello

strcat(str1, str2): HelloWorld

strlen(str1): 10

**Good practice when using C-style strings:**

* Use **secured variants** (e.g. strcat\_s() or strncat() instead of strcat()) to avoid buffer overflow (and warning C4996). [Why](https://www.geeksforgeeks.org/strcat-vs-strncat-c/)? [And here](https://en.cppreference.com/w/c/string/byte/strcat#Notes)?

### Strings as Character Pointers

A string created by a character pointer can be stored in two ways:

**1) Read-only string in a shared segment (of the** [**Initialized Data Segment**](#_Data_Segment_(Initialized)**)**

When string value is directly assigned to a pointer, in most of the compilers, it’s stored in a **read-only** block.

In the following program, "GfG" is stored in a shared read-only block, but pointer str is stored in a read-write block. **You can change str to point something else but CANNOT change value at present str**. So, this kind of stringshould only be used when we don’t want to modify the stringat a later stage in program.

int main() {

char\* str = "GfG"; // Stored in read-only part of data segment

\*(str+1) = 'n'; // Segmentation Fault: trying to modify read-only memory

// Or str[1] = 'n'; // [Why](http://www.cplusplus.com/forum/beginner/126853/#msg686764)?

return 0;

}

**Note 1:** The above program works fine in C, but in C++ the compiler will show warning: “*deprecated conversion from string constant to char\**”. That’s because in C, string literals are arrays of char, but in C++ they are constant array of char. Therefore, we should use const keyword before char\*.

const char\* str = "GfG"; // valid in C++, invalid in C

**Note 2:**

* const char\* ptr: A pointer to a constant character. You cannot change the value pointed by ptr, but you can change ptr itself.
* char\* const ptr: A constant pointer to non-constant character. You cannot change ptr, but can change the value pointed by ptr.

**2) Dynamically allocated in Heap segment**

This way, the string is stored in a **read-write** memory and can be modified at a later stage in the program.

char\* str;

int size = 4; // one extra for '\0'

str = (char\*)malloc(sizeof(char)\*size);

\*(str+0) = 'G';

\*(str+1) = 'f';

\*(str+2) = 'G';

\*(str+3) = '\0';

**Convert between char\* and char[]**

|  |  |  |
| --- | --- | --- |
|  | **Can’t** | **Can** |
| const char\* src = "abc";  char\* src = "abc";  🡪  char dest[10]; | dest = src;  dest = (char)src; | memcpy(dest, src, strlen(src));  memcpy(dest, src, 10); |
| char src[] = "abc";  🡪  char\* dest;  const char\* dest; |  | dest = src; |

**Pass char\* by references:**

In following examples, we want to change value of outA and outB after executed setAB(). So the correct output will be: a is 2 and b is abc

This is WRONG:

#include <iostream>

using namespace std;

void setAB(int\* outA, const char\* outB) // in C, can use "char\* outB"

{

\*outA = 2;

// If "outB = (char\*)malloc(sizeof(char)\*10);", output of outB will be still "abc"

outB = "xyz";

// since outB is a pointer to a constant char, you can change outB ifself,

// but cannot change the value it points to

}

int main()

{

int a = 1;

const char\* b = "abc"; // in C, can use "char\* b = "abc""

// if "const char\* b = (char\*)calloc(10, sizeof(char)); b = "abc";", output of outB will be "abc"

setAB(&a, b);

cout << "a is " << a << " and b is " << b;

}

Output:

a is 2 and b is abc

This is CORRECT:

#include <iostream>

using namespace std;

void setAB(int\* outA, const char\*\* outB) // in C, can use "char\*\* outB"

{

\*outA = 2;

\*outB = "xyz";

}

int main()

{

int a = 1;

const char\* b = "abc"; // in C, can use "char\* b = "abc""

setAB(&a, &b);

cout << "a is " << a << " and b is " << b;

}

Output:

a is 2 and b is xyz

This is CORRECT:

#include <iostream>

using namespace std;

void setAB(int\* outA, const char\*& outB) // in C, can use "char\*& outB"

{

\*outA = 2;

outB = "xyz";

}

int main()

{

int a = 1;

const char\* b = "abc"; // in C, can use "char\* outB"

// If "const char\* b = (char\*)calloc(10, sizeof(char));", output of outB will be "xyz"

setAB(&a, b);

cout << "a is " << a << " and b is " << b;

}

Output:

a is 2 and b is xyz

This is CORRECT:

#include <iostream>

using namespace std;

void setAB(int\* outA, char\* outB)

{

\*outA = 2;

strcpy\_s(outB, sizeof(char)\*10, "xyz");

// if using "outB = (char\*)"xyz";", output of outB will be "abc"

}

int main()

{

int a = 1;

char\* b = (char\*)calloc(10, sizeof(char));

b = (char\*)"abc";

setAB(&a, b);

cout << "a is " << a << " and b is " << b;

}

Output:

a is 2 and b is xyz

## String Standard Library

This library internally uses char array to store character, but **all memory management, allocation and null termination are handled by** [**string class**](http://www.cplusplus.com/reference/string/) **itself** - why it is easy to use.

As string class is a container class, we can iterate over all its characters using an iterator similar to other containers (vector, set, map, etc.). But generally, we use a simple for loop for iterating over the characters and index them using []operator.

**Note: size (length) vs capacity**

Capacity is the maximum number of characters that the string can currently hold without having to grow. It is created when allocating memory.

Size (length) is the number of characters that actually exist in the string.

The following example uses pre-built operators in C++ string class to support copy and concatenation of strings:

#include <iostream>

#include <string>

using namespace std;

int main() {

   string str1 = "Hello";

   string str2 = "World";

   string str3;

   int  len ;

   // copy str1 into str3

   str3 = str1;

   cout << "str3: " << str3 << endl;

   // concatenates str1 and str2

   str3 = str1 + str2;

   cout << "str1 + str2: " << str3 << endl;

   // total length of str3 after concatenation

   len = str3.size();

   cout << "str3.size(): " << len << endl;

   return 0;

}

Output:

str3: Hello

str1 + str2: HelloWorld

str3.size(): 10

**So, char\* vs std::string – which one should be prefered?**

When dealing exclusively in C++, std:string is the best way to go because of better searching, replacement, and manipulation functions.

But there are some cases where you might prefer char\* over std:string.

* Compatible with C code (although std::string::c\_str() handles type-casting).
* Conserve memory (std::string will likely have more overhead).

## sprintf, snprintf, sprintf\_s …

These functions are used to specifically **write formatted data to a string:**

|  |  |
| --- | --- |
| **Functions** | **Parameters** |
| int sprintf(  char\* buffer,  const char\* format,  ...); | * buffer − The pointer to an array of char elements where the **output** string is stored. * format − The input string to be written to thebuffer. It optionally contains format specifiers (starting with %) which are replaced by the values of variables passed to the function as additional arguments. These format specifiers follow the same specifications as [of printf](http://www.cplusplus.com/printf). * ... – Depending on the number of format strings, the function may expect additional arguments; each containing a value to be used to replace a format specifier in the format string. * *Return value* – On success, the number of characters that would have been written, not counting the terminating null character. On failure, a negative number is returned. |
| int snprintf(  char\* buffer,  size\_t length,  const char\* format,  ...); | * length – The maximum number of **bytes** to be used in the buffer.   Difference between sprintf and snprintf:   * sprintf provides NO bounds checking on the buffer’s size. The buffer will be overrun if its size is smaller than the format. * snprintf prevents [buffer overrun](#_2bn6wsx). If its size is too small for the text being printed, it is set to an empty string and the invalid parameter handler is invoked. In the general case, snprintf() is more secure than sprintf. |
| [Non-standard]  int sprintf\_s(  char\* buffer,  size\_t length,  const char\* format,  ...); | Difference between sprintf\_s and snprintf: The first one guarantees that the buffer will be null-terminated (unless its size is zero). |
| [Windows-only]  int \_snprintf\_s(  char\* buffer,  size\_t length,  size\_t count,  const char\* format  ...  ); | * count: Maximum number of **characters** to be used in the buffer, or [\_TRUNCATE](https://docs.microsoft.com/en-us/cpp/c-runtime-library/truncate?view=vs-2019) (which enables truncation behavior). |
| [Windows-only]  template <size\_t size>  int \_snprintf\_s(  char (&buffer)[size],  size\_t count,  const char\* format  ...); // C++ only |  |
| int swprintf(  wchar\_t\* buffer,  size\_t length,  const wchar\_t\* format,  ...);  int swprintf\_s(...);  int \_snwprintf\_s(...); | swprintf, swprintf\_s and \_snwprintf\_s are wide-character versions of sprintf, sprintf\_s and \_snprintf\_s, respectively. |

**Example: snprintf**

#include <stdio.h>

int main() {

char buffer[50];

int a = 5;

double b = 3.2;

int retSize = snprintf(buffer, sizeof(buffer), "%d plus %.2f is %.2f", a, b, a+b);

printf("[%s] is a string with %d chars long\n", buffer, retSize);

return 0;

}

Output:

[5 plus 3.20 is 8.20] is a string with 19 chars long

**sprintf() vs stringstream - Which one should be prefered?**

They do different things. The fact that you can use std::stringstream to replicate some of the functionality of sprintf does not mean it's just a C++ rewrite of sprintf.

Which one should be prefered really depends. In general:

**When sprint() and its variants:**

1. Where performance is a big consideration, sprintf() might give you a performance boost over using std::stringstream.

2. sprintf() is easier to work with, particularly when writing specially-formatted strings. Sure, you can do this with std::stringstream modifiers, but it's lengthy; you can't see what the code is achieving at a glance.

3. sprintf() is easier to work with if you're passing a char[] buffer to a C API. If you use std::stringsteam, you have to do type-casting from std::stringstream to std::string to C-style string (using ssObj.str().c\_str()).

**When std::stringstream:**

1. The standard library for streams supports much more than sprintf() does.

2. It's meant to be extensible and flexible. It's also object-oriented, which means you can override the standard class to get the behavior you want.

3. std::stringstream operations often use growable buffers internally; which implies relatively slow memory allocations.

## \_splitpath\_s, \_wsplitpath\_s

Breaks a path name into components.

<https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/splitpath-s-wsplitpath-s?view=vs-2019>

## ANSI and Unicode

A character can be represented in 1 byte or 2 bytes. A 1-byte character is **ANSI** character - all English characters are represented through this encoding. A 2-byte character is **Unicode**, which can represent ALL languages in the world.

The C++ compiler supports char and wchar\_t as native data-types for ANSI and Unicode characters, respectively.

### char vs wchar\_t vs TCHAR

What if you want your C/C++ code to be independent of character encoding used? Here is the solution:

**Way 1:**

Replace:

**char** cResponse;

Same as:

**CHAR** cResponse;

with

**wchar\_t** cResponse; // Wide characters

Same as:

**WCHAR** cResponse;

**Way 2:**

In order to support multi-lingual (i.e., Unicode), you can simply code it in **more generic way**:

#include<TCHAR.H>

TCHAR cResponse;

// In TCHAR.H

#ifdef \_UNICODE

typedef **wchar\_t** **TCHAR**;

#else

typedef **char** **TCHAR**;

#endif

So:

* If your project is being compiled as Unicode (the macro \_UNICODE is defined), TCHAR would translate to wchar\_t.
* If it is being compiled as ANSI, it would be translated to char.

You are free to use char and wchar\_t, and project settings will not affect any direct use of these keywords.

In Visual Studio, the macro \_UNICODE will be defined when you set Character Set to "*Use Unicode Character Set*". Otherwise, you set Character Set if set to "*Use Multi-Byte Character Set*".

### Prefix \_tc in Function Names

Likewise, to support multiple character sets using single code base, apply macros to functions. Instead of using strcpy(), strlen(), strcat() – including the secure versions suffixed with \_s; or wcscpy(), wcslen(), wcscat() – including secure, you should better use use \_tcscpy(), \_tcslen(), \_tcscat().

That’s because:

#ifdef \_UNICODE

#define \_tcslen wcslen

#else

#define \_tcslen strlen

#endif

You should refer TCHAR.H to lookup more macro definitions like this.

### Suffix A and W in Function Names

// WinUser.H

#ifdef UNICODE

#define SetWindowText SetWindowTextW

#else

#define SetWindowText SetWindowTextA

#endif

### Prefix L and \_T in String Text

"This is ANSI String. Each letter takes 1 byte." // ANSI

L"This is Unicode string. Each letter would take 2 bytes." // Unicode

\_T("Either string, depending on compilation."); // ANSI or Unicode

That’s because:

#ifdef \_UNICODE

#define \_T(c) L##c

#define TEXT(c) L##c

#else

#define \_T(c) c

#define TEXT(c) c

#endif

### LPSTR vs LPWSTR vs LPCTSTR …

The meaning goes like:

* LP – Long Pointer
* C – Constant
* STR – String
* WSTR – Wide-character String
* T – TCHAR

That’s because:

// #include <Windows.h>

typedef char\* LPSTR // long pointer to string

typedef const char\* LPCSTR; // long pointer to constant string

typedef WCHAR\* LPWSTR; // long pointer to wide-character string

typedef const WCHAR\* LPCWSTR; // long pointer to constant wide-character string

#ifdef \_UNICODE

#define LPCTSTR  LPCWSTR

#else

#define LPCTSTR  LPCSTR

#endif

### When Type-Casting Won’t Work!

For example:

// Wrong code

int main()

{

TCHAR name[] = "Saturn";

strlen(name);

}

On ANSI build, this code will successfully compile. But in Unicode, let’s say UNICODE/\_UNICODE is defined (i.e. "Use Unicode Character Set" in project settings), the compiler would report set of errors:

*error C2440: 'initializing' : cannot convert from 'const char [7]' to 'TCHAR []'*

*error C2664: 'strlen' : cannot convert parameter 1 from 'TCHAR []' to 'const char \*'*

And the programmers would start committing mistakes by correcting it following ways:

// 1. The conversion is not possible from TCHAR\* to TCHAR[7].

TCHAR name[] = (TCHAR\*)"Saturn";

// 2. Cannot convert parameter 1 from 'const char [7]' to 'const wchar\_t \*'

wcslen("Saturn");

Unfortunately, this error can be incorrectly corrected by simple C-style typecast:

strlen((const char\*)name);

or

wcslen((const wchar\_t\*)"Saturn");

And you'd think you've attained one more experience level in pointers! You are wrong - the code would give incorrect result, and in most cases would simply cause Access Violation. Typecasting this way is like passing a float variable where a structure of 80 bytes is expected (logically).

The string "Saturn" is sequence of 7 bytes:

'S' (83) 'a' (97) 't' (116) 'u' (117) 'r' (114) 'n' (110) '\0' (0)

But when you pass same set of bytes to wcslen, it treats each 2-byte as a single character. Therefore, first two bytes [97, 83] would be treated as one character having value: 24915 (97<<8 | 83). It is Unicode character: ?. And the next character is represented by [117, 116] and so on.

For sure, you didn't pass those set of Chinese characters, but improper typecasting has done it! Therefore, it is very essential to know that type-casting will not work!

So, the following is **what you must do**:

// Correct code

TCHAR name[] = \_T("Saturn");

\_tcslen(\_T("Saturn"));

or

wcslen(L"Saturn");

**In short, typecasting will not work**. You either need to represent strings in correct form itself, or use ANSI to Unicode, and vice-versa, routines for conversions.

## QAs

### Why there is null character for each string?

The last element of the character array is reserved for the null character ('\0'), which serves as the string's termination marker. It **indicates the end of the string and distinguishes it from other character sequences**.

It's important to note that the null character is not part of the actual string content.

### What is string interning?

"String interning" (sometimes called "string pooling") is an **optimization technique in which only one copy of a string is stored**, no matter how many times the program references it. It involves creating a single shared instance of each unique string and ensuring that all references to the same string point to that shared instance.

**How string interning works:**

1. When a string is encountered, its contents are checked against the existing pool of interned strings.

* If a matching string already exists in the pool, a reference to the existing interned string is returned.
* If the string is not found in the pool, a new interned string is created by allocating memory and storing the string's contents.

1. The reference to the interned string is returned, and subsequent references to the same string will also point to the interned instance.

**Benefits of string Iinterning:**

* **Reduced memory usage**: Because duplicate copies of the same string content are eliminated, memory is saving significantly, especially when dealing with large numbers of strings or when similar strings are used frequently.
* **Faster string comparison**: Because interned strings have unique instances, string comparison operations can be optimized by comparing references instead of comparing the actual string contents (character-by-character). This is helpful when string comparisons are frequent, such as sorting or searching operations.
* **Improved Caching and Hashing**: Once a string is interned, its hash value or cached results (if any) can be reused. This avoids the need for recalculation or duplication of caching structures.

**Implementation:**

In C++, string interning is **not a built-in feature** of the standard library. However, you can implement it as following:

#include <iostream>

#include <unordered\_map>

#include <string>

class StringIntern {

private:

    std::unordered\_map<std::string, std::string> internedStrings;

public:

    const std::string& Intern(const std::string& str) {

        auto it = internedStrings.find(str);

        if (it != internedStrings.end()) {

            return it->second;   // Return existing interned string

        } else {

            internedStrings[str] = str;  // Intern the new string and return it

            return internedStrings[str];

        }

    }

};

int main() {

    StringIntern stringIntern;

    const std::string& str1 = stringIntern.Intern("Hello");

    const std::string& str2 = stringIntern.Intern("World");

    const std::string& str3 = stringIntern.Intern("Hello");

    std::cout << "str1: " << str1 << std::endl;  // Output: Hello

    std::cout << "str2: " << str2 << std::endl;  // Output: World

    std::cout << "str3: " << str3 << std::endl;  // Output: Hello

    // Check if the interned strings share the same memory address

    std::cout << "str1 address: " << &str1 << std::endl;

    std::cout << "str2 address: " << &str2 << std::endl;

    std::cout << "str3 address: " << &str3 << std::endl;

    return 0;

}

Output:

str1: Hello

str2: World

str3: Hello

str1 address: 0x557c644c6ed8

str2 address: 0x557c644c6fa8

str3 address: 0x557c644c6ed

You can see that the str1 and str3 share the same memory address because both are Hello.

# Array

An array is a data structure which stores a fixed-size **sequential collection of variables of the same type**.

Instead of declaring individual variables, such as number0, number1, ... to number99, you should declare one array variable such as numbers[100] and use numbers[0], numbers[1], ... to numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

## Declaring Arrays

Here is the syntax to declare a single-dimension array.

type arrayName [arraySize];

Where

* arraySize: the number of elements required by that array; must be an integer constant greater than zero.
* type: the type of the elements; can be any valid C/C++ data type.

Example:

double balance[10]; // 10-element array called 'balance' of type double

## Initializing Arrays

The number of values between braces {} cannot be larger than the number of elements that we declare for the array between square brackets [].

You can initialize C++ array elements either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0}; // good

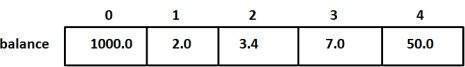
Or

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0}; // best

Or initialize all elements as 0.0

double balance[] = {0.0};

The pictorial representation of this array is:



## Accessing Elements

An element is accessed by **indexing the array name**. This is done by placing the index of the element within brackets [] after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable.

Example:

#include <iostream>

using namespace std;

#include <iomanip>

using std::setw;

int main() {

int n[10]; // n is an array of 10 integers

// initialize elements of array n to 0

for ( int i = 0; i < 10; i++ ) {

n[i] = i + 100; // set element at location i to i + 100

}

cout << "Element" << setw(13) << "Value" << endl; // setw: format the output

// output each array element's value

for ( int j = 0; j < 10; j++ ) {

cout << setw(7)<< j << setw(13) << n[j] << endl;

}

return 0;

}

Output:

Element Value

0 100

1 101

2 102

3 103

4 104

5 105

6 106

7 107

8 108

9 109

## Dynamic Array

In above sections, we talked about *static array*. Now let’s move to *dynamic array*.

Once a static array has been created, its size cannot be changed. However, a dynamic array can expand its size even after it has been filled. Note that its elements also occupy a contiguous block of memory.

**Example:**

In C++, we can create a dynamic array using the new keyword or malloc function. The number of items to be allocated is specified within a pair of bracket [] or ().

#include<iostream>

using namespace std;

int main() {

    int x, n;

    cout << "Enter the number of items: ";

    cin >> n;

    int\* arr = new int[n]; // dynamic array

    cout << "Enter " << n << " items: " << endl;

    for (x = 0; x < n; x++) {

        cin >> arr[x];

    }

    cout << "You entered: ";

    for (x = 0; x < n; x++) {

        cout << arr[x] << " ";

    }

delete[] arr; // free memory of dynamic array

    return 0;

}

Output:

Enter the number of items: 3

Enter 3 items:

1

2

3

You entered: 1 2 3

**Resizing dynamic arrays:**

The length of a dynamic array is set during the allocation time. However, C++ **doesn’t have a built-in mechanism** of resizing an array once it has been allocated.

You can overcome this challenge by allocating a new array dynamically, copying over the elements, then erasing the old array.

For example:

#include <iostream>

using namespace std;

int main() {

    // Create a dynamic array with initial size

    int initialSize = 5;

    int\* dynamicArray = new int[initialSize];

    // Initialize the array with some values

    for (int i = 0; i < initialSize; i++) {

        dynamicArray[i] = i + 1;

    }

    // Display the initial array

    cout << "Initial Array: ";

    for (int i = 0; i < initialSize; i++) {

        cout << dynamicArray[i] << " ";

    }

    cout << endl;

    // Create a new temporary array with the desired size

    int newSize = 8;

    int\* tempArray = new int[newSize];

    // Copy elements from the original array to the temporary array

    for (int i = 0; i < initialSize; i++) {

        tempArray[i] = dynamicArray[i];

    }

    // Release the memory for the original array

    delete[] dynamicArray;

    // Point the dynamicArray pointer to the new resized array

    dynamicArray = tempArray;

    // Display the resized array

    cout << "Resized Array: ";

    for (int i = 0; i < newSize; i++) {

        cout << dynamicArray[i] << " ";

    }

    cout << endl;

    // Don't forget to deallocate the memory for the final dynamic array

    delete[] dynamicArray;

    return 0;

}

Output:

Initial Array: 1 2 3 4 5

Resized Array: 1 2 3 4 5 0 0 0

## Important Array Concepts

|  |  |
| --- | --- |
| **Concept** | **Description** |
| Multi-dimensional arrays | The simplest form of MD arrays is the two-dimensional array. |
| Pointer to an array | You can generate a pointer to the first element of an array by simply specifying the array name, without any index. |
| Passing arrays to functions | You can pass to the function a pointer to an array by specifying the array's name without an index. |
| Return array from functions | C++ allows a function to return an array. |

### Multi-Dimensional Arrays

Here is the syntax to declare a multi-dimensional array:

type name[size1][size2]...[sizeN];

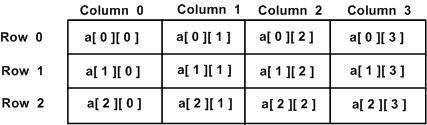
#### Two-Dimensional Arrays

The simplest form of MD arrays is the two-dimensional array. A 2D array is, in essence, a list of one-dimensional arrays. To declare a 2D integer array of size x, y, you would write something as follows:

type arrayName[x][y];

Where type can be any valid C++ data type and arrayName will be a valid C++ identifier.

**A 2D can be seen as a table**, which will have x number of rows and y number of columns. A 2-dimensional array a[3][4], which contains three rows and four columns can be shown as below:



Thus, every element in array *a* is identified by an element name of the form a[i][j], where *a* is the name of the array, and *i* and *j* are the subscripts that uniquely identify each element in *a*.

##### Initializing 2D Arrays

MD arrays may be initialized by specifying braces {} value for each row. Following is an array with 3 rows, each row have 4 columns.

int a[3][4] = {

{0, 1, 2, 3} , // initializers for row indexed by 0

{4, 5, 6, 7} , // initializers for row indexed by 1

{8, 9, 10, 11} // initializers for row indexed by 2

};

##### Accessing 2D-Array Elements

An element in 2D array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

int val = a[2][3];

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above program.

#include <iostream>

using namespace std;

int main() {

// an array with 5 rows and 2 columns.

int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

for ( int i = 0; i < 5; i++ )

for ( int j = 0; j < 2; j++ ) {

cout << "a[" << i << "][" << j << "]: ";

cout << a[i][j]<< endl;

}

return 0;

}

Output:

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

### Pointer to Array

**An array name is a constant** [**pointer**](#_lnxbz9) **to the first element of the array**. Therefore, in the declaration:

double balance[50];

balance is a pointer to &balance[0]*,* which is the address of the first element of the array balance. Thus:

double \*p;

double balance[10];

p = balance; // Same as &balance[0]

It’s legal to use array names as constant pointers, and vice versa. Therefore, \*(balance+4) is a legal way of accessing the data at balance[4]*.* Once you store the address of first element in p, you can access array elements using \*p, \*(p+1), \*(p+2) and so on.

For example:

#include <iostream>

using namespace std;

int main() {

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

double \*p;

p = balance;

cout << "Array values using pointer " << endl;

for (int i = 0; i < 5; i++) {

cout << "\*(p + " << i << "): ";

cout << \*(p + i) << endl;

}

cout << endl << "Array values using balance as address " << endl;

for (int i = 0; i < 5; i++) {

cout << "\*(balance + " << i << "): ";

cout << \*(balance + i) << endl;

}

return 0;

}

Output:

Array values using pointer

\*(p + 0): 1000

\*(p + 1): 2

\*(p + 2): 3.4

\*(p + 3): 17

\*(p + 4): 50

Array values using balance as address

\*(balance + 0): 1000

\*(balance + 1): 2

\*(balance + 2): 3.4

\*(balance + 3): 17

\*(balance + 4): 50

In the above example, p is a pointer to double which means it can store address of a variable of double type. Once we have address in p, then \*p will give us value available at the address stored in p, as we have shown in the above example.

### Passing Arrays to Functions

C++ does NOT allow to pass an ENTIRE array as an argument to a function (only the first element of the array is passed). However, you can pass a pointer to the first element of the array by specifying the array's name without an index.

Here is a WRONG example to pass an entire array as an argument to a function:

void func(int inArr[]) {

size\_t size = sizeof(inArr);

printf("%d\n", size); // Output: 8

}

void main() {

int arr[] = {1, 2, 3};

size\_t size = sizeof(arr); // assume int = 4 bytes

printf("%d\n", size); // Output: 12

func(arr);

}

*You might not know!*

The loss of type and dimensions of an array is known as **Array Decay**. It occurs when we pass the array into a function by pointer or value. First address is sent to the array which is a pointer. That is why, the size of array is not the original one.

In practice, if you want to pass a 1D array as an argument in a function, you would have to declare function parameter in one of following ways.

**Way 1: Parameters as a pointer with the array size** (to the first element of the array)

void myFunction(int\* param, int size) {

...

}

**Way 2: Parameters as a sized array with the array size**

void myFunction(int param[10], int size) {

...

}

**Way 3: Parameters as an un-sized array with the array size**

void myFunction(int param[], int size) {

...

}

**Way 4: Pass array to function by reference**

void myFunction(int (&param)[3]) { // suppose: 3 is array size

...

}

For example:

#include <iostream>

using namespace std;

double getAverage(int arr[], int size) {

int i, sum = 0;

double avg;

for (i = 0; i < size; ++i) {

sum += arr[i];

}

avg = double(sum) / size;

return avg;

}

int main() {

int balance[] = {1, 2, 3, 4, 5};

double avg;

avg = getAverage(balance, 5) ;

cout << "Average value is: " << avg << endl;

return 0;

}

Output:

Average value is: 3

### Return Array from Functions

C++ does NOT allow to return an ENTIRE array to outside of the function. However, you can return a pointer to the first element of the array by specifying the array's name without an index.

int\* myFunction() {

...

}

In practice, there are following ways:

**Way 1: Define the local array as static** (reason: memory will be keep outside of the function, so prevent the pointer from being 'dangling')

**Way 2: Use dynamic array** (reason: memory will be keep outside of the function, so prevent the pointer from being 'dangling')

For example:

// generate 5 random numbers

#include <iostream>

#include <ctime>

using namespace std;

int\* getRandom() {

static int r[5]; // way 1

// int\* r = new int[5]; // way 2

// set the seed

srand((unsigned)time( NULL ));

for (int i = 0; i < 5; ++i) {

r[i] = rand();

cout << r[i] << endl;

}

return r;

}

int main() {

int \*p;

p = getRandom();

for (int i = 0; i < 5; i++) {

cout << "\*(p + " << i << "): ";

cout << \*(p + i) << endl;

}

// delete p; // if way 2

return 0;

}

Output:

624723190

1468735695

807113585

976495677

613357504

\*(p + 0): 624723190

\*(p + 1): 1468735695

\*(p + 2): 807113585

\*(p + 3): 976495677

\*(p + 4): 613357504

## QAs

### Array vs Pointer

|  |  |
| --- | --- |
| **Array** | **Pointer** |
| A data structure storing a **group of elements** of the same type. | A variable storing **address of a single variable**, function or object. |
| Memory is allocated in a **contiguous** block, either **statically** or dynamically. | Memory is allocated **dynamically** allocated. Memory allocation is **random**. |
| Support indexing using either **square brackets** or **pointer arithmetic**. | Support indexing using **pointer arithmetic**. |

### Array vs Vector

|  |  |
| --- | --- |
| **Arrays** | **Vector** |
| Array are contiguous memory locations of homogenous data types stored in a **fixed** **size**. | Vectors have **dynamic size** and can be resized at runtime. Elements can be added or removed easily. |
| Offer **limited functionality** and rely on manual algorithms for operations like sorting or searching. | Provide **various buit-in functions** and algorithms in the <vector> header, STL library, such as sorting, searching, inserting, and erasing elements, as well as exception handling. |
| Require using an **index** to access elements. | Provide either **index** or **iterator** to access elements. |
| Require **deallocating memory** after used. | If vector of objects (not pointer), automatically delete the used memory. |

### Array vs List

|  |  |
| --- | --- |
| **Arrays** | **Lists** |
| Array are **contiguous** memory locations of homogenous data types stored in a **fixed** **size**. | Lists are individual elements that are **linked** to each other with the help of pointers and have **flexible size**. Elements can be added or removed easily. |
| Uses less memory than linked lists. | Uses more memory as it has to store the value and the pointer memory location. |
| Require using an **index** to access elements. | Require using an **iterator** to access elements. |

### How are MD arrays implemented in memory?

MD arrays are implemented using a technique called "row-major order" or "row-major layout" which determines how elements are stored in contiguous memory.

In particular, elements are arranged in memory by rows. The elements of the first row are stored first, followed by the elements of the second row, and so on. Within each row, the elements are stored contiguously.

Here's an example:

int arr[3][4] = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12}

};

In memory, the elements of this 2D array would be stored as follows:

1 2 3 4 5 6 7 8 9 10 11 12

To access an element in a MD array, the compiler uses pointer arithmetic and the knowledge of the size of each dimension to calculate the correct memory address. For example, to access arr[2][3], the compiler calculates the memory address as (baseAddress + (2\*numCol + 3) \* sizeof(int)), where numCol represents the number of columns in the array.

### Ragged and Jagged Arrays

### Sparse Array

# Pointer

Some C++ tasks are performed more easily with pointers. And other C++ tasks, such as [linked list](#_qsh70q) or [dynamic memory allocation](#_4i7ojhp), cannot be performed without them.

## Memory Address

Every variable is a memory location, and every memory location has its **address** defined which can be accessed using & operator. Consider the following example which will print the address of the variables defined:

#include <iostream>

using namespace std;

int main() {

int var1;

char var2[10];

cout << "Address of var1 variable: ";

cout << &var1 << endl;

cout << "Address of var2 variable: ";

cout << &var2 << endl;

return 0;

}

Output:

Address of var1 variable: 0xbfebd5c0

Address of var2 variable: 0xbfebd5b6

## What Are Pointers?

**A pointer is a variable whose value is the address of another variable**. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

type \*var\_name;

For example:

int \*ip; // pointer to an integer

double \*dp; // pointer to a double

float \*fp; // pointer to a float

char \*ch; // pointer to character

The actual data type of the value of all pointers, whether *integer*, *float*, *character*, is the SAME: a **long hexadecimal number** (4 bytes in 32-bit system, or 8 bytes in 64-bit system) that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

## Access Values Stored in Pointers

To access the value at the address available in the pointer variable, we use an **operator \* that returns the value of the variable located at the address** specified by its operand.

#include <iostream>

using namespace std;

int main() {

int var = 20; // actual variable declaration

int \*ip; // pointer variable

ip = &var; // store address of var in pointer variable

cout << "Value of var variable: ";

cout << var << endl;

// print the address stored in ip pointer variable

cout << "Address stored in ip variable: ";

cout << ip << endl;

// access the value at the address available in pointer

cout << "Value of \*ip variable: ";

cout << \*ip << endl;

return 0;

}

Output:

Value of var variable: 20

Address stored in ip variable: 0xbfc601ac

Value of \*ip variable: 20

## Important Pointer Concepts in C++

Pointers have many concepts. The following is few important ones:

|  |  |
| --- | --- |
| **Concept** | **Description** |
| Null Pointers | This is a pointer that points to nowhere. |
| Void Pointers | This is a special type of pointer that can hold the address of any type of data but lacks type information. |
| Pointer Arithmetic | There are 4 arithmetic operators that can be used on pointers: ++, --, +, - |
| Pointers vs Arrays | There is a close relationship between pointers and arrays. |
| Array of Pointers | You can define arrays to hold a number of pointers. |
| Pointer to Pointer | C++ allows you to have pointer on a pointer and so on. |
| Passing Pointers to Functions | Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function. |
| Return Pointer from Functions | C++ allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well. |
| The this Pointer | This is a special poiter to all member functions of class. |

### Null Pointer

It is always a **good practice to assign NULL to a pointer** (called a null pointer**)** when declaring it if you don’t have exact address to be assigned. By avoiding the use of a null pointer, you can avoid the accidental misuse of an uninitialized pointer. Many times, uninitialized variables hold some junk values and it becomes difficult to debug the program.

Consider the following program:

#include <iostream>

using namespace std;

int main() {

int\* ptr = NULL;

cout << "The value of ptr is " << ptr ;

return 0;

}

Output:

The value of ptr is 0

On most OSs, programs are not permitted to access memory at address 0 because that memory is reserved by the OS. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, **if a pointer contains the null value, it is assumed to point to nothing**.

To check for a null pointer, you can use an if statement as follows:

if(ptr) // succeeds if p is not null

if(!ptr) // succeeds if p is null

#### *nullptr* vs *NULL*

NULL is a macro which is actually an integer (0) assigned to a pointer because of an implicit conversion.

nullptr is a new keyword in C++11, representing a value of self-defined type that can convert into a pointer, but not into integers.

int i = NULL; // ok

int i = nullptr; // error - not an integer convertible value

int\* p = NULL; // ok - int converted into pointer

int\* p = nullptr; // ok (should use)

Suppose you have two functions in overload:

1. void func(int x);

2. void func(int\* x);

Now, if you call func(NULL), you are actually calling the first variant. But func(nullptr) will call the second one.

To avoid the risk to call one function instead of another, always use 0 if you want an integer, and nullptr if you want a pointer. The use of NULL, therefore, should be avoided.

### Void Pointer

In C and C++, a void pointer is a special type of pointer that can **hold the address of any type** of data but lacks type information. It is a flexible mechanism that allows you to work with memory addresses without specifying the specific data type it points to.

Example:

#include <iostream>

using namespace std;

void print(void\* p) {

    int\* iP = static\_cast<int\*>(p);    // type casting

    cout << \*iP << endl;               // Output: 42

}

int main() {

    int value = 42;

    void\* p = &value;

    print(p);

    return 0;

}

In above example, we use static\_cast to convert the void\* to an int\* pointer, allowing us to access the value it points to. So note that when working with void pointer, you need to be cautious about type safety and ensure that you **correctly cast it to the appropriate type** before dereferencing it. Improper usage may lead to undefined behavior or runtime errors.

### Pointer Arithmetic

Pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are 4 arithmetic operators for pointers: ++, --, +, and -

To understand, let consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let perform the following arithmetic operation on the pointer:

ptr++;

The ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer. This operation will move the pointer to next memory location without impacting actual value at the memory location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

#### Incrementing a Pointer

We **prefer using a pointer instead of an array because the variable pointer can be incremented, unlike the array which cannot be incremented because it is a constant pointer.** The following program increments the variable pointer to access each succeeding element of the array:

#include <iostream>

using namespace std;

const int MAX = 3;

int main() {

int var[MAX] = {10, 100, 200};

int\* ptr;

// let us have array address in pointer.

ptr = var; // No need & here because array is actually a pointer

for (int i = 0; i < MAX; i++) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the next location

ptr++;

}

return 0;

}

Output:

Address of var[0] = 0xbfa088b0

Value of var[0] = 10

Address of var[1] = 0xbfa088b4

Value of var[1] = 100

Address of var[2] = 0xbfa088b8

Value of var[2] = 200

#### Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

#include <iostream>

using namespace std;

const int MAX = 3;

int main() {

int var[MAX] = {10, 100, 200};

int\* ptr;

// let us have address of the last element in pointer.

ptr = &var[MAX-1]; // But need & here

for (int i = MAX; i > 0; i--) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the previous location

ptr--;

}

return 0;

}

Output:

Address of var[3] = 0xbfdb70f8

Value of var[3] = 200

Address of var[2] = 0xbfdb70f4

Value of var[2] = 100

Address of var[1] = 0xbfdb70f0

Value of var[1] = 10

#### Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1].

#include <iostream>

using namespace std;

const int MAX = 3;

int main() {

int var[MAX] = {10, 100, 200};

int\* ptr; // point to the first element of the array

// let us have address of the first element in pointer.

ptr = var;

int i = 0;

while (ptr <= &var[MAX-1]) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the previous location

ptr++;

i++;

}

return 0;

}

Output:

Address of var[0] = 0xbfce42d0

Value of var[0] = 10

Address of var[1] = 0xbfce42d4

Value of var[1] = 100

Address of var[2] = 0xbfce42d8

Value of var[2] = 200

### Pointers vs Arrays

Pointers and arrays are strongly related. In fact, **pointers and arrays are interchangeable in many cases**. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing.

For example:

#include <iostream>

using namespace std;

const int MAX = 3;

int main() {

int var[MAX] = {10, 100, 200};

int\* ptr;

// let us have array address in pointer

ptr = var;

for (int i = 0; i < MAX; i++) {

cout << "Address of var[" << i << "] = ";

cout << ptr << endl;

cout << "Value of var[" << i << "] = ";

cout << \*ptr << endl;

// point to the next location

ptr++;

}

return 0;

}

Output:

Address of var[0] = 0xbfa088b0

Value of var[0] = 10

Address of var[1] = 0xbfa088b4

Value of var[1] = 100

Address of var[2] = 0xbfa088b8

Value of var[2] = 200

However, pointers and arrays are not completely interchangeable. For example:

#include <iostream>

using namespace std;

const int MAX = 3;

int main() {

int var[MAX] = {10, 100, 200};

for (int i = 0; i < MAX; i++) {

\*var = i; // This is a correct syntax

var++; // This is incorrect

}

return 0;

}

It is perfectly acceptable to apply the pointer operator \* to var but it is illegal to modify var value. The reason for this is that var is a constant that points to the beginning of an array and cannot be used as l-value.

Because an array name generates a pointer constant, it can still be used in pointer-style expressions, as long as it is not modified. For example, the following is a valid statement that assigns var[2] the value 500:

\*(var + 2) = 500;

Above statement is valid and will compile successfully because var is not changed.

### Array of Pointers

Before we understand the concept of array of pointers, let us consider the following example, which makes use of an array of 3 integers:

#include <iostream>

using namespace std;

const int MAX = 3;

int main() {

int var[MAX] = {10, 100, 200};

for (int i = 0; i < MAX; i++) {

cout << "Value of var[" << i << "] = ";

cout << var[i] << endl;

}

return 0;

}

Output:

Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200

There may be a situation, when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

int \*ptr[MAX];

This declares ptr as an array of MAX-integer pointers. Thus, each element in ptr now holds a pointer to an int value. Following example makes use of 3 integers which will be stored in an array of pointers:

#include <iostream>

using namespace std;

const int MAX = 3;

int main() {

int var[MAX] = {10, 100, 200};

int\* ptr[MAX];

for (int i = 0; i < MAX; i++) {

ptr[i] = &var[i]; // assign the address of integer.

}

for (int i = 0; i < MAX; i++) {

cout << "Value of var[" << i << "] = ";

cout << \*ptr[i] << endl;

}

return 0;

}

Output:

Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200

You can also use an array of pointers to character to store a list of strings as follows:

#include <iostream>

using namespace std;

const int MAX = 4;

int main() {

const char\* names[MAX] = { "Zara Ali", "Hina Ali", "Nuha Ali", "Sara Ali" };

for (int i = 0; i < MAX; i++) {

cout << "Value of names[" << i << "] = ";

cout << (names + i) << endl;

}

return 0;

}

Output:

Value of names[0] = 0x7ffd256683c0

Value of names[1] = 0x7ffd256683c8

Value of names[2] = 0x7ffd256683d0

Value of names[3] = 0x7ffd256683d8

### Pointer to Pointer

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared by placing an additional \* operator in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int:

int \*\*var;

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the \* operator be applied twice, as is shown below in the example:

#include <iostream>

using namespace std;

int main() {

int var;

int \*ptr;

int \*\*pptr;

var = 3000;

// take the address of var

ptr = &var;

// take the address of ptr using address of operator &

pptr = &ptr;

// take the value using pptr

cout << "Value of var: " << var << endl;

cout << "Value available at \*ptr: " << \*ptr << endl;

cout << "Value available at \*\*pptr: " << \*\*pptr << endl;

return 0;

}

Output:

Value of var: 3000

Value available at \*ptr: 3000

Value available at \*\*pptr: 3000

### Passing Pointers to Functions

To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

#include <iostream>

#include <ctime>

using namespace std;

void getSeconds(unsigned long \*par);

int main() {

unsigned long sec;

getSeconds(&sec);

// print the actual value

cout << "Number of seconds: " << sec << endl;

return 0;

}

void getSeconds(unsigned long \*par) {

// get the current number of seconds

\*par = time(NULL);

return;

}

Output:

Number of seconds: 1294450468

The function which can accepts a pointer, can also accept an array as shown in the following example:

#include <iostream>

using namespace std;

double getAverage(int \*arr, int size);

int main() {

// an int array with 5 elements

int balance[5] = {1, 2, 3, 2, 3};

double avg;

// pass pointer to the array as an argument

avg = getAverage( balance, 5 ) ;

// output the returned value

cout << "Average value is: " << avg << endl;

return 0;

}

double getAverage(int \*arr, int size) {

int i, sum = 0;

double avg;

for (i = 0; i < size; ++i) {

sum += arr[i];

}

avg = double(sum) / size;

return avg;

}

Output:

Average value is: 3

### Return Pointer from Functions

C++ allows to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example:

int \*myFunction() {

...

}

Same example as [here](#_3rdcrjn).

### The "this" Pointer

Every object in C++ has **access to its own address** through an important pointer called this pointer. It is an implicit **poiter to all member functions of class**. Therefore, inside a member function, this may be used to refer to the invoking object.

Note: Friend functions do not have this pointer, because friends are not members of a class.

For example:

#include <iostream>

using namespace std;

class Box {

public:

// Constructor definition

Box(double l = 2.0, double b = 2.0, double h = 2.0) {

cout << "Constructor called." << endl;

length = l;

width = b;

height = h;

}

double Volume() {

return length \* width \* height;

}

int compare(Box box) {

return this->Volume() > box.Volume();

}

private:

double length; // Length of a box

double width; // Width of a box

double height; // Height of a box

};

int main(void) {

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

if(Box1.compare(Box2)) {

cout << "Box2 is smaller than Box1" << endl;

} else {

cout << "Box2 is equal to or larger than Box1" << endl;

}

return 0;

}

Output:

Constructor called.

Constructor called.

Box2 is equal to or larger than Box1

## QAs

### What is Dangling Pointer?

"Dangling pointers" are pointers that **point to a memory location has been deallocated or freed** which is no longer valid or owned by the program. Dereferencing a dangling pointer can lead to undefined behavior, data corruption or even crash.

**Causes:**

Dangling pointers typically occur in following situations:

* After memory region pointed by the pointer is **deallocated** using delete or free(), that pointer becomes danging.
* When a pointer is declared within a local scope (e.g., inside a function) and then accessed outside that scope. Once the scope is exited, the pointer becomes invalid.
* When using pointer arithmetic incorrectly. For example, incrementing or decrementing a pointer beyond the bounds of an allocated memory block.

Example:

#include <stdio.h>

int\* danglingPointer() {

    int temp = 10;

    return &temp;

}

int main() {

    int \*ptr = danglingPointer();

    // ptr is a dangling pointer now

    printf("%d", \*ptr);

    return 0;

}

Compilation Warning :

address of local variable 'temp' returned [-Wreturn-local-addr]

int temp = 10;

^~~~

Output :

812249860 <-- garbage value

### What is Pointer Aliasing?

Pointer aliasing refers to the case where **two or more pointers of different types access the same memory location**. In other words, pointer aliasing occurs when multiple pointers can potentially refer to overlapping memory regions. This violate the *Strict Aliasing Rules* which states that an object should only be accessed through a pointer of a compatible type. This eventually **leads to** **unexpected behavior**.

For example:

#include <iostream>

using namespace std;

int main() {

    int x = 1;

    int\* ptr1 = &x;

    char\* ptr2 = reinterpret\_cast<char\*>(ptr1);

    \*ptr2 = 'a';

    cout << "x: " << x << endl;     // Output: 97

    return 0;

}

In above example, we modified an int object through a char\* pointer and violated the *Strict Aliasing Rules*.

To prevent this issue, we can use the restrict keyword in C and the \_\_restrict qualifier in C++ to inform the compiler that a pointer does not alias with any other pointer.

# Reference

**A reference variable is an alias, that is, another name for an already existing variable**. Think of a variable name as a *label* attached to the variable's location in memory, then you can think of a reference as a *second label* attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference.

## Reference vs Pointer

Reference variables are often confused with pointer variable, but 3 major differences between references and pointers are:

* You CANNOT have a NULL reference. You must always be able to assume that a reference variable is connected to a legal piece of storage.
* A reference variable MUST BE INITIALIZED when it is created. By contrast, a pointer can be initialized at any time.
* Once a reference variable is initialized to an object, it CANNOT BE CHANGED to refer to another object. By contrast, a pointer can be pointed to another object at any time.
* The memory location of reference can be accessed easily or it can be used directly. By contrast, the memory location of a pointer cannot be accessed easily as we **have to use a dereference ‘ \* ‘** .

**Note**: Reference is a new concept introduced in C++. There is no reference in C.

## Creating References

Example 1: With a variable

#include <iostream>

using namespace std;

int main() {

// declare a simple variable

int i;

// declare a reference variable

int& r = i;

i = 5;

cout << "Value of i: " << i << endl;

cout << "Value of i reference: " << r << endl;

r = 1;

cout << "Value of i: " << i << endl;

cout << "Value of i reference: " << r << endl;

return 0;

}

Output:

Value of i: 5

Value of i reference: 5

Value of i: 1

Value of i reference: 1

Example 2: With an array

#include <iostream>

using namespace std;

int main() {

int arr[5] = {1, 2, 3, 4, 5};

// Declare a reference to the array

int (&ref)[5] = arr;

// Access and modify array elements through the reference

ref[2] = 10;

cout << arr[2];

return 0;

}

Output:

10

## Using References

### Function Parameters

We have discussed how we implement **call by reference** concept using pointers. Here is another example of call by reference which makes use of C++ reference:

#include <iostream>

using namespace std;

// function definition to swap the values

void swap(int& x, int& y) {

int temp;

temp = x; // save the value at address x

x = y; // put y into x

y = temp; // put x into y

return;

}

int main() {

// local variable declaration

int a = 1;

int b = 2;

cout << "Before swap, value of a: " << a << endl;

cout << "Before swap, value of b: " << b << endl;

// calling a function to swap the values

swap(a, b);

cout << "After swap, value of a: " << a << endl;

cout << "After swap, value of b: " << b << endl;

return 0;

}

Output:

Before swap, value of a: 1

Before swap, value of b: 2

After swap, value of a: 2

After swap, value of b: 1

### Function Return Value

A C++ function can return a reference in a similar way as it returns a pointer.

**When a function returns a reference, it returns an implicit pointer to its return value**. This way, a function can be used on the LEFT SIDE of an assignment statement. For example:

#include <iostream>

#include <ctime>

using namespace std;

int vals[] = {1, 2, 3}; // global variable

int& setValues(int i) {

return vals[i];

}

int main() {

cout << "Value before change:" << endl;

for (int i = 0; i < 3; i++) {

cout << "vals[" << i << "] = ";

cout << vals[i] << endl;

}

setValues(1) = 20; // change 2nd element

setValues(2) = 30; // change 3th element

cout << "Value after change:" << endl;

for (int i = 0; i < 3; i++) {

cout << "vals[" << i << "] = ";

cout << vals[i] << endl;

}

return 0;

}

Output:

Value before change:

vals[0] = 1

vals[1] = 2

vals[2] = 3

Value after change:

vals[0] = 1

vals[1] = 20

vals[2] = 30

Note:

When returning a reference, be careful that the object being referred to DOES NOT go out of scope. In other words, it is illegal to return a reference to local variable. But you can always return a reference on a **static** variable.

int& func() {

// int q;

// return q; // Compile time error

static int x;

return x; // Safe, x lives outside this scope

}

int a = func();

### QAs

#### Advantages of References over Pointers

* Function parameters:
  + More convenient: **simpler syntax**
  + Safer: **cannot be null** and must be initialized, so don’t need to check for null pointers or handle potential null pointer exceptions.
  + Efficiency: unlike passing by value, passing by reference **eliminates the need for creating a new copy of the object**, which can save both time and memory.
* Function return value:
  + Efficiency: unlike returning by value, returning by reference **eliminates the need for creating a new copy of the object**, which can save both time and memory.
  + Enabling *function chaining*: multiple function calls can be chained together in a single expression. This can lead to more readable code.

Example:

class MyVector {

public:

MyVector& add(int value) {

// Add 'value' to the vector

// ...

return \*this; // Return a reference to the current object

}

MyVector& multiply(int factor) {

// Multiply each element of the vector by 'factor'

// ...

return \*this; // Return a reference to the current object

}

};

int main() {

MyVector v;

v.add(5).multiply(2); // Function chaining

return 0;

}

# Input / Output

## C-Style

### 'scanf()' and 'printf()'

In C, the built-in functions:

* scanf() is used to **read formatted input from the standard input** (such as keyboard).
* printf() is used to **send formatted output to the console**.

Both the functions are defined in in stdio.h.

Depending on the data type of the variable presented in the scanf() or printf(), we have to use the corresponding *format string* to get the correct output.

The following table lists format strings for common data type.

%s cannot be used for std::string

|  |  |
| --- | --- |
| **Data type** | **Format string** |
| int | %d |
| char | %c |
| string  (char[] or char\*) | %s |
| float | %f |
| double | %lf |
| short int | %hd |
| unsigned int | %u |
| long int | %li |
| long long int | %lli |
| unsigned long int | %lu |
| unsigned long long int | %llu |
| signed char | %c |
| unsigned char | %c |
| long double | %Lf |

**Example:**

#include <stdio.h>  // Stands for "Standard input output"

int main() {

    // Print "Hello World!" with a following newline

    printf("Hello World!\n");

    // Input an integer, then print its value

    int a;

    printf("Enter a: ");

    scanf("%d", &a);  // Note the & symbol

    printf("a = %d\n", a);

    // Input a floating number, then print its value

    float b;

    printf("Enter b: ");

    scanf("%f", &b);

    printf("b = %.2f\n", b); // %.2f means "only get 2 decimal places"

    // Input a character, then print its value

    char c;

    printf("Enter c: ");

    scanf(" %c", &c); // Note a space before %d (reason [here](https://stackoverflow.com/a/13543113))

    printf("c = %c\n", c);

// Print a string (Way 1)

char d[] = {'a', 'b', 'c', '\0'}; // Same as "abc"

printf("%s", d);

// Print a string (Way 2)

char d[] = {'a', 'b', 'c'};

printf("%.\*s", (int)sizeof(d), d);

// Note: The precision after the . specifies the maximum number of characters

// to output from the string.

// It can be given as a decimal number, or as \*,

// and provided as an int argument before the char pointer.

    return 0;

}

Output:

Hello World!

Enter a: 13

a = 13

Enter b: 3.1456

b = 3.15

Enter c: z

c = z

**Tips:**

* The printf() returns the number of characters printed by it, and the scanf() returns the number of characters read by it. For example:

int i = printf("studytonight");

The above statement will return 12 as result, which will be stored in the variable i.

* We can take multiple inputs with scanf(). For example:

scanf("%d/%d/%d", &month, &day, &year);

The above state will return 5/14/2010 if you enter "5/14/2010".

**Note:**

* The **scanf() stops reading characters when it encounters a space**. To solve this problem, we have two ways:
  + Use %[^\n]%\*c inside scanf(), instead of %s. Here, [] is the scanset character. ^\n tells to take input until newline doesn’t get encountered. Then, with this %\*c, it reads newline character and here used \* indicates that this newline character is discarded.

Another pattern: %[^\n]s

* + Use [gets()](#_'gets()'_and_'puts()') function.
* There is a problem with scanf() when there is a scanf() or fgets()/gets() after it, or when using scanf() in a loop. For example:

#include<stdio.h>

int main() {

   int x;

   printf("Enter x: ");

   scanf("%d", &x);

   char str[100];

   printf("Enter str: ");

   gets(str);

   printf("x = %d, str = %s\n", x, str);

   printf("End");

   return 0;

}

Output:

Enter x: 10

Enter str: x = 10, str =

End

As you can see from the output, the scanf() reads an integer and **leaves a newline character in buffer**. So fgets() only reads newline and the input for str is ignored.

The similar problem occurs when scanf() is used in a loop.

There is one way to solve this problem: **Add a** [getchar()](#_'getchar()'_and_'putchar()') **after scanf() to read an extra newline**.

### 'gets()' and 'puts()'

* gets() reads a line from stdin (standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs.
* puts() writes the string str and a trailing newline to stdout.

**Example**:

#include<stdio.h>

void main() {

    // character array of length 100

    char str[100];

    printf("Enter a string: ");

    gets(str);

    puts(str);

    printf("End");

    getch();

}

Output:

Enter a string: ho nhan tri

ho nhan tri

End

**Note**:

* The gets() and puts() are **only used with string**. They cannot work with other data types.
* If the scanf() stops reading characters when it encounters a space, **the gets() reads space as character** too. This is why we want to use gets() to input long string with spaces.
* The **gets() is deprecated** in new versions of C (because it can cause buffer overflow). So, if using C99 or earlier, never use gets(), use fgets() rather. If using C11 use gets\_s(). Example on [fgets()](https://www.geeksforgeeks.org/fgets-gets-c-language/).

### 'getchar()' and 'putchar()'

* getchar() reads a character from the terminal and returns it as an integer.
* putchar() displays the character passed to it on the screen and returns the same character.

These functions read/display only single character at a time. You can use this method in a loop in case you want to read/display more than one character.

**Example**:

#include <stdio.h>

void main() {

    int c;

    printf("Enter a character: ");

    // Take a character as input and store it in variable c

    c = getchar();

    // Display the character stored in variable c

    putchar(c);

}

Output:

Enter a character: abc

a

## C++-Style

Using std::cin and std::cout

# Header File

## What Are Header Files?

Header files **contain declarations of functions, variables, macros, and other elements** that you want to use in your program. They act like blueprints or interfaces, allowing you to **share these declarations across multiple source files** without duplicating code. This makes your code more organized, modular, and easier to maintain.

That's said header files (e.g., .h files) typically (and should) contain only declarations, while the corresponding definitions are placed in source files (e.g., .c or.cpp files).

## Example

Our header file is math\_operations.h:

int add(int a, int b);

int subtract(int a, int b);

Our source file is math\_operations.cpp:

#include "math\_operations.h"

#include <iostream>

int add(int a, int b) {

int sum = a + b;

std::cout << "Sum is " << sum << std::endl;

return sum;

}

int subtract(int a, int b) {

return a - b;

}

And our main.c which uses functions in math\_operations:

#include "math\_operations.h"

int main() {

int result = add(5, 3); // Calls the add function

// Do something with result

return 0;

}

By putting the declaration of add() and subtract() functions in the header file, we can reuse these function in multiple source files by simply including math\_operations.h.

Notes:

* Use the #include directive following by the header file (relative path) to include a header file.
* The **compiler inserts the header's content into your source code** at this point. So you should put all **header files at the top of source files**.
* Use **double quotes** for user-defined headers: #include "math\_operations.h"
* Use **angle brackets** for standard library headers: #include <iostream>

## File Extensions

Header files use specific extensions to distinguish them from source files:

|  |  |
| --- | --- |
| C | Typically use the .h extension (e.g., myheader.h).  Note: Standard library headers, like <stdio.h>, have extension. |
| C++ | Often use .hpp to indicate C++-specific headers (e.g., myheader.hpp), though .h is still common and valid.  You might see .hh or .hxx, but these are less standard and mostly a matter of preference or project convention.  Note: Standard library headers, like <iostream> or <vector>, have no extension. |

The choice of extension is a convention, not a strict rule — the compiler doesn’t care as long as the file is included correctly.

## Best Practices

To use header files effectively and avoid common pitfalls, follow these best practices:

### Use Include Guards

They help prevent a header from being included multiple times (which can cause errors like duplicate declarations) .

Simply wrap the header content with #ifndef, #define, and #endif:

#ifndef MATH\_OPERATIONS\_H

#define MATH\_OPERATIONS\_H

int add(int a, int b);

int subtract(int a, int b);

#endif // MATH\_OPERATIONS\_H

Alternatively, in C++, you can use #pragma once compiler directive (not standard, but supported by most modern compilers).

### Declare Only, Don't Define

Keep header files for declarations only (e.g., function prototypes). Put definitions in source files to avoid linker errors (e.g., multiple definition of…).

Exception: In C++, **inline functions and templates can be defined in headers** since they need to be available at compile time.

### Minimize Dependencies

* Use **forward declarations** when possible. This tells the compiler that the implementation class exists without providing its full definition.

For example, instead of "#include <mystruct.h>", should "class MyStruct;".

* **Include only what you need** in headers.

For example, if mycode.h only needs a type from types.h, include types.h instead of a broader header like all\_utils.h.’

* **Include in implementation** **files** when possible.

For example, if mycode.h does not need TypeA from types.h, but mycode.cpp calls TypeA, then include types.h in mycode.cpp only.

If both mycode.h and mycode.cpp calls TypeA, then technically you can include types.h in mycode.h only. But it's recommended to explicitly include types.h in both mycode.h and mycode.cpp.

* Consider PIMPL (pointer to implementation) for complex classes.

### Avoid Circular Dependencies

If header1.h includes header2.h, and header2.h includes header1.h, you’ll get compilation errors.

Instead, use forward declarations or redesign to break the cycle.

### Organization

Include headers in the following order:

* Associated header file (in .cpp files)
* C++ standard library headers
* Other library headers
* Your project's headers

Example:

// In foo.cpp

#include "foo.h" // Associated header first

#include <string> // Standard library headers

#include <vector>

#include <third\_party.h> // Third-party libraries

#include "project/bar.h" // Project headers

### Consider Precompiled Headers for Large Projects

Advanced projects might use precompiled headers to speed up compilation.

# Pre-Increment and Post-Increment

## Pre-Increment Operator vs Post-Increment Operator

Suppose, x = ++i then

i = 0;

i = i + 1; // i will be incremented by 1 first

x = i; // x = 1

Suppose, x = i++ then

i = 0;

x = i // x = 0

i = i + 1; // then i will be incremented by 1

++i increases the value of i then returns its new value, whereas i++ returns the previous value of i then increases its value.

## Performance

* **For pre-defined data types** (like int, long): Both operations take same time and do the same operation on i with only difference in the return value. So, **use whichever you prefer**.
* **For user-defined data types** (like iterators): **Use** **pre-increment (++i) whenever you can** for better performance. Reason:
  + With ++i, the increment operation is performed first, and then the updated value is returned. This mean no temporary copy of i is required.
  + With i++, the increment operation is performed, but the original value of i is returned before the increment. This requires creating a temporary copy of the original value, which adds an extra step.

Examples:

Both

for(int i = 0; i < 4; ++i) {

printf("%d", i);

}

and

for(int i = 0; i < 4; i++) {

printf("%d", i);

}

produce the same results and take same time.

The same thing happens for

int i = 0;

while(i < 4) {

printf("%d", i);

i++;

}

and

int i = 0;

while(i < 4) {

printf("%d", i);

++i;

}

But

int i = 0;

int j = i;

while(j < 4) {

printf("%d ", i); // output: 0 1 2 3

j = ++i;

}

differs from:

int i = 0;

int j = i;

while(j < 4) {

printf("%d ", i); // output: 0 1 2 3 4

j = i++;

}

Or

vector<int> v(n);

for(vector<int>::iterator it = v.begin(); it < v.end(); ++it)

has better performance than:

vector<int> v(n);

for(vector<int>::iterator it = v.begin(); it < v.end(); it++)

# File Stream

This tutorial will teach you how to read and write from a file. This requires a standard C++ library called <fstream>, which defines three new data types:

|  |  |
| --- | --- |
| **Data Type**  **(Class)** | **Description** |
| ofstream | Represents the output file stream. Used to **create** files and **write** info to files. |
| ifstream | Represents the input file stream. Used to **read** info from files. |
| fstream | Has the capabilities of **both** ofstream and ifstream. |

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

## Opening a File

A file must be opened before you can read from it or write to it. Either ofstream object may be used to open a file for writing purpose and ifstream for reading purpose.

To open a file, use the open() function:

void fstream::open(const char \*filename, ios::openmode mode);

* filename: name and location of the file to be opened
* mode: mode in which the file should be opened.

|  |  |
| --- | --- |
| **Mode Flag** | **Description** |
| std::ios::app | Append mode. All output to that file to be appended to the end. |
| std::ios::ate | Open a file for output and move the read/write control to the end of the file. |
| std::ios::in | Open a file for reading. |
| std::ios::out | Open a file for writing. **If the file has not been created, create it.** |
| std::ios::trunc | If the file already exists, its content will be truncated before opening the file. |

You can combine two or more of these values by OR-ing them together. For example, if you want to open a file in write mode and truncate it in case that already exists, following will be the syntax:

ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc);

Similar way, you can open a file for reading and writing purpose as follows:

fstream afile;

afile.open("file.dat", ios::out | ios::in);

## Closing a File

When a C++ program terminates, it automatically flushes all the streams, release all the allocated memory and close all the opened files. But **it is always a good practice that a programmer should close all the opened files before program termination**.

To close an opening file, use close() function:

void fstream::close();

## Reading from a File

To read from a file, use the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the std::cin object.

**Note**: The >> operator will stop reading until it finds a space or a newline. To deal with spaces, use the getline() function.

## Writing to a File

To write data to a file, use the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the std::cout object.

**Note**: The << operator will stop writing until it finds a space or a newline. To deal with spaces, use the getline() function.

## Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named *input.txt*, the program reads information from the file and outputs it back onto the screen:

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

int main() {

   string data;

   // =============== Write to file ================ //

   // Open a file in write mode

   ofstream writeFile;

   writeFile.open("input.txt");

   if (!writeFile) {

      cerr << "Open file error" << endl;

return -1;

   }

   // Get user input

   cout << "Writing to the file" << endl;

   cout << "Enter your name: ";

   // Copy data from 'cin' to data (including white space, tab, and eof)

   getline(cin, data);

   // Write inputed data into the file

   writeFile << data << endl;

   // Again get user input

   cout << "Enter your age: ";

   cin >> data;

   // Again write inputted data into the file

   writeFile << data << endl;

   // Close the opened file.

   writeFile.close();

   // =============== Read from file ================ //

   // Open a file in read mode

   ifstream readFile;

   readFile.open("input.txt");

   if (!readFile) {

      cerr << "Open file error" << endl;

return -1;

   }

   // Read the file

   cout << endl << "Your file's content is: " << endl;

   // To read white space, tab and eof in the file,

   // Use getline() instead of readFile >> data;

   getline(readFile, data);

   // Display the data

   cout << data << endl;

   // Again read the data from the file and display it

   readFile >> data;

   cout << data << endl;

   // Close the opened file

   readFile.close();

   return 0;

}

Output:

Writing to the file

Enter your name: Ho Nhan Tri

Enter your birthday: 11/10/1995

Your file's content is:

Ho Nhan Tri

11/10/1995

## File Position Pointers

Both istream and ostream provide member functions for repositioning the file-position pointer. These member functions are seekg ("seek get") for istream and seekp ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be ios::beg (the default) for positioning relative to the beginning of a stream, ios::cur for positioning relative to the current position in a stream or ios::end for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

// position to the nth byte of fileObject (assumes ios::beg)

fileObject.seekg(n);    // n = 0 --> position to the beginning of the file

// position n bytes forward in fileObject

fileObject.seekg(n, ios::cur);

// position n bytes back from end of fileObject

fileObject.seekg(n, ios::end);

// position at end of fileObject

fileObject.seekg(0, ios::end);

# Structure

**Structure is a user-defined data type which allows you to combine data items of different kinds.**

## Defining Structures

The struct statement defines a new data type, with more than one member, for your program:

struct [structure tag] {

member definition;

member definition;

...

member definition;

} [one or more structure variables];

Where:

* *structure tag*: optional.
* *member definition*: a normal variable definition, such as int i, float f or any other valid variable definition.
* *one or more structure variables*: optional.

Example of the Book structure:

struct Books {

char title[50];

char author[50];

int book\_id;

} book;

## Define Structures with 'typedef'

There is an easier way to define structs or you could "alias" types you create. For example:

typedef struct \_Books {

char title[50];

char author[50];

int book\_id;

} Books;

Now, you can use Books directly to define variables of Books type without using struct keyword. Following is the example:

Books Book1, Book2;

## Create Instances of Structures

**Without 'typedef':**

Way 1:

#include <iostream>

using namespace std;

struct Student {

string name;

int id;

} student; // Note: 'student' is an instance

int main()

{

student.name = "abc";

student.id = 1;

cout << "Student name: " << student.name <<endl;

cout << "Student ID: " << student.id << endl;

cout << "Size of struct: " << sizeof(student) << endl;

}

Way 2:

#include <iostream>

using namespace std;

struct Student {

string name;

int id;

};

int main()

{

struct Student student; // Note: Still work if do not using "struct"

student.name = "abc";

student.id = 1;

cout << "Student name: " << student.name <<endl;

cout << "Student ID: " << student.id << endl;

cout << "Size of struct: " << sizeof(student) << endl;

}

**With 'typedef':**

Way 1:

#include <iostream>

using namespace std;

typedef struct \_Student {

string name;

int id;

} Student;

int main()

{

struct Student student; // Note: Syntax error if not using "struct"

student.name = "abc";

student.id = 1;

cout << "Student name: " << student.name <<endl;

cout << "Student ID: " << student.id << endl;

cout << "Size of struct: " << sizeof(struct Student) << endl;

}

Way 2:

#include <iostream>

using namespace std;

typedef struct \_Student {

string name;

int id;

};

int main()

{

\_Student student; // Note: Syntax error if using "struct"

student.name = "abc";

student.id = 1;

cout << "Student name: " << student.name <<endl;

cout << "Student ID: " << student.id << endl;

cout << "Size of struct: " << sizeof(\_Student) << endl;

}

## Accessing Structure Members

To access any member of a structure, use the **member access operator (.)**.

#include <iostream>

#include <cstring>

using namespace std;

struct Books

{

char title[50];

char author[50];

int book\_id;

};

int main()

{

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// book 1 specification

strcpy(Book1.title, "Learn C++ Programming");

strcpy(Book1.author, "Chand Miyan");

Book1.book\_id = 6495407;

// book 2 specification

strcpy(Book2.title, "Telecom Billing");

strcpy(Book2.author, "Yakit Singha");

Book2.book\_id = 6495700;

// Print Book1 info

cout << "Book 1 title : " << Book1.title <<endl;

cout << "Book 1 author : " << Book1.author <<endl;

cout << "Book 1 id : " << Book1.book\_id <<endl;

// Print Book2 info

cout << "Book 2 title : " << Book2.title <<endl;

cout << "Book 2 author : " << Book2.author <<endl;

cout << "Book 2 id : " << Book2.book\_id <<endl;

return 0;

}

Output:

Book 1 title : Learn C++ Programming

Book 1 author : Chand Miyan

Book 1 id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Yakit Singha

Book 2 id : 6495700

## Initialize Structures

Way 1:

#include <iostream>

using namespace std;

typedef struct \_Student {

string name;

int id;

} Student;

int main()

{

// In C99, it's called designated initializer

Student student = { .name = "abc", .id = 1}; // Items don't need to be in order

// Another way

// Student student = { "abc", 1}; // Items must be in order

// Note: Without typedef, these two ways still work.

cout << "Student name: " << student.name <<endl;

cout << "Student ID: " << student.id << endl;

}

Way 2:

#include <iostream>

using namespace std;

struct \_Student {

string name;

int id;

\_Student(string n, int i) // struct constructor

{

name = n;

id = i;

}

};

int main()

{

Student student("abc", 1);

cout << "Student name: " << student.name <<endl;

cout << "Student ID: " << student.id << endl;

}

## Structure as Function Argument

You can pass a structure as a function argument in very similar way as you pass any other variable.

#include <iostream>

#include <cstring>

using namespace std;

struct Books {

char title[50];

char author[50];

int book\_id;

};

void printBook(struct Books book)

{

cout << "Book title : " << book.title <<endl;

cout << "Book author : " << book.author <<endl;

cout << "Book id : " << book.book\_id <<endl;

}

int main()

{

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// book 1 specification

strcpy(Book1.title, "Learn C++ Programming");

strcpy(Book1.author, "Chand Miyan");

Book1.book\_id = 6495407;

// book 2 specification

strcpy(Book2.title, "Telecom Billing");

strcpy(Book2.author, "Yakit Singha");

Book2.book\_id = 6495700;

// Print Book1 info

printBook(Book1);

// Print Book2 info

printBook(Book2);

return 0;

}

Output:

Book title : Learn C++ Programming

Book author : Chand Miyan

Book id : 6495407

Book title : Telecom Billing

Book author : Yakit Singha

Book id : 6495700

## Pointers to Structure

You can define pointers to structures in very similar way as you define pointer to any other variable.

struct Books \*pBooks;

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

pBooks = &Book1;

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

pBooks ->title; // same as (\*pBooks).title;

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

#include <iostream>

#include <cstring>

using namespace std;

struct Books {

char title[50];

char author[50];

int book\_id;

};

// This function accept pointer to structure as parameter.

void printBook(struct Books \*book)

{

cout << "Book title : " << book->title <<endl;

cout << "Book author : " << book->author <<endl;

cout << "Book id : " << book->book\_id <<endl;

}

int main()

{

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// Book 1 specification

strcpy(Book1.title, "Learn C++ Programming");

strcpy(Book1.author, "Chand Miyan");

Book1.book\_id = 6495407;

// Book 2 specification

strcpy(Book2.title, "Telecom Billing");

strcpy(Book2.author, "Yakit Singha");

Book2.book\_id = 6495700;

// Print Book1 info, passing address of structure

printBook(&Book1);

// Print Book1 info, passing address of structure

printBook(&Book2);

return 0;

}

Output:

Book title : Learn C++ Programming

Book author : Chand Miyan

Book id : 6495407

Book title : Telecom Billing

Book author : Yakit Singha

Book id : 6495700

## Structure with Constructor

Way 1:

struct Foo {

int bar;

int boo;

Foo(int inBar, int inBoo) :

bar(inBar),

boo(inBoo)

{};

};

The code above declares a struct called Foo with two members: bar and boo. This struct has a constructor which takes inBar and inBoo as initializing values for bar and boo, respectively.

Way 2:

struct Foo {

int bar;

int boo;

Foo(int inBar, int inBoo)

{

bar = inBar;

boo = inBoo;

}

};

When creating an instance of this struct:

Foo sFoo(1, 2) // bar = 1 and boo = 2.

## QAs

### Struct vs Class in C++

|  |  |
| --- | --- |
| **Struct** | **Class** |
| Members of the struct are always by default **public** mode. | Members of the class can be in **private, protected, and public** modes. |
| Structure does **not support inheritance**. | Classes support the concept of **inheritance**. |
|  |  |

# Enum

An enumeration is used for **defining named constant values**. It is declared using the enumkeyword.

## Syntax

The simplest form of an enum definition is as follows:

enum enum\_name {

enumeration list

};

Each of the elements in the *enumeration list* stands for an **integer value** in an increasing order of **+1**. By default, the value of the first element is **0**. But we can change this valueby assigning it to a specific value.

For example:

#include<stdio.h>

// Now this enum starts at 1, not 0

enum YEAR {

JAN = 1, FEB, MAR, APR, MAY, JUN,

JUL, AUG, SEP, OCT, NOV, DEC

};

int main() {

for (int i = JAN; i <= DEC; i++) {

printf("%d ", i);

}

return 0;

}

Output:

1 2 3 4 5 6 7 8 9 10 11 12

## Notes

1.Allenum values must be integral numbers. So we can't have an enum such as { PI=3.14, E=2.71 }.

2. Two enum element can have the same value. In fact, we can assign any integral number to the enum list*.* For example:

enum STATE {FAILED = 0, FREEZED = 0, WORKING, BROKEN = -1};

3.All enum elements must be unique in their scope. For example:

enum STATE {WORKING, FAILED};

enum RESULT {FAILED, PASSED}; // Wrong syntax because dupliate FAILED

## Enum vs Macro

**Why enum:**

1. Enums follow scope rules.

2. Enum variables are automatically assigned values. Following is simpler and easier to read:

enum DAY {MONDAY, TUESDAY, WEDNESDAY};

In this case, defining names constants using macro takes more lines and harder to read:

#define MONDAY 0

#define TUESDAY 1

#define WEDNESDAY 2

3. A macro is a preprocessor thing, and the compiled code has no idea about the identifiers you create. They have been already replaced by the preprocessor before the code hits the compiler. An enum is a compile time entity, and the compiled code retains full information about the symbol, which is available in the debugger (and other tools). So, enum identifiers propagate into the debugging information.

**Why macro:**

1. Enums only work with integers. On the other hand, macros can be of any type. They can even be any code block containing statements, loops, function calls, etc.

2. Enums are actually variables, so they require memory to be stored.

## Scoped Enums (C++11)

Conventional enums in C++98 export their elements in a global scope, which can lead to following issues:

1. Two enum cannot share the same names.
2. No variable can have a name which is already in an enum.
3. Enums are not type-safe.

#include <bits/stdc++.h>

int main() {

enum Gender { Male, Female };

enum Color { Red, Green };

Gender gender = Male;

Color color = Red;

if (gender == color) { // warning by compiler

std::cout << "Equal";

}

return 0;

}

For the first two issues, we can resolve them by putting a prefix before the enum name and each of its elements to make them distinct. But for the last issue, it’s an unpleasant coding experience.

So, C++11 introduces the enum class keyword to prevent exporting enumerators in the surrounding scope. Moreover, we can also inherit from an enum.

Example:

#include <iostream>

int main() {

enum class Color { Red, Green, Blue };

// Variable can have the same name as an enum class element

int Green = 10;

// Instantiate an enum class

Color x = Color::Green;

// Comparison now is completely type-safe

if (x == Color::Green) {

std::cout << "It's Green\n";

} else {

std::cout << "It's not Green\n";

}

// won't work as there is no implicit conversion to int

// std::cout << x;

std::cout << int(x);

return 0;

}

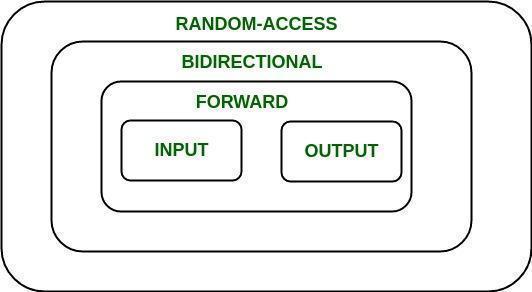
# STL Iterator

## Definition

An iterator is an object (similar to a pointer) that **points to an element inside the container**. We can use iterators to move through the contents of the container to access a particular content at a particular location.

Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. **The most obvious form of iterator is a pointer** which can point to elements in an array, and can iterate through them using the increment operator (++). But all iterators do not have similar functionality as that of pointers.

Depending upon the functionality of iterators, they can be classified into five categories, as shown in the diagram below with the outer one being the most powerful, and consequently the inner one is the least powerful in terms of functionality.



Note that different containers support different iterators, as given below:

|  |  |
| --- | --- |
| **Container** | **Types of Iterator Supported** |
| Vector | Random-Access |
| List | Biodirectional |
| Deque | Random-Access |
| Map | Biodirectional |
| Multimap | Biodirectional |
| Set | Biodirectional |
| Multiset | Biodirectional |
| Stack | No iterator supported |
| Queue | No iterator supported |
| Priority Queue | No iterator supported |

## Types of Iterators

Based upon the functionality of the iterators, they can be classified into five major categories:

1. [**Input Iterators**](https://www.geeksforgeeks.org/input-iterators-in-cpp/): They are the weakest type of iterators with very limited functionality. They can only be used in a single-pass algorithm, i.e., those algorithms which process the container sequentially such that no element is accessed more than once.
2. [**Output Iterators**](https://www.geeksforgeeks.org/output-iterators-cpp/): Just like input iterators, they are very limited in functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.
3. [**Forward Iterator**](https://www.geeksforgeeks.org/forward-iterators-in-cpp/): They are higher in hierarchy than input and output iterators, and contain all the features present in these two iterators. But as the name suggests, they can only move in forward direction and that one step at a time.
4. [**Bidirectional Iterators**](https://www.geeksforgeeks.org/bidirectional-iterators-in-cpp/): They have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators, as they can move in both directions.
5. [**Random-Access Iterators**](https://www.geeksforgeeks.org/random-access-iterators-in-cpp/): They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality is same as pointers.

The properties of each iterator category are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Category** | | | | **Properties** | **Valid expressions** |
| All categories | | | | [*copy-constructible*](http://www.cplusplus.com/CopyConstructible), [*copy-assignable*](http://www.cplusplus.com/CopyAssignable)*and*[*destructible*](http://www.cplusplus.com/Destructible) | X b(a); b = a; |
| Can be incremented | ++a a++ |
| [Random Access](http://www.cplusplus.com/RandomAccessIterator) | [Bidirectional](http://www.cplusplus.com/BidirectionalIterator) | [Forward](http://www.cplusplus.com/ForwardIterator) | [Input](http://www.cplusplus.com/InputIterator) | Supports equality/inequality comparisons | a == b a != b |
| Can be dereferenced as an *rvalue* | \*a a->m |
| [Output](http://www.cplusplus.com/OutputIterator) | Can be dereferenced as an *lvalue*  (only for *mutable iterator types*) | \*a = t \*a++ = t |
|  | [*default-constructible*](http://www.cplusplus.com/DefaultConstructible) | X a; X() |
| Multi-pass: neither dereferencing nor incrementing affects dereferenceability | { b=a; \*a++; \*b; } |
|  | | Can be decremented | --a a-- \*a-- |
|  | | | Supports arithmetic operators + and - | a + n n + a a - n a - b |
| Supports inequality comparisons (<, >, <= and >=) between iterators | a < b a > b a <= b a >= b |
| Supports compound assignment operations += and -= | a += n a -= n |
| Supports offset dereference operator ([]) | a[n] |
| *X: iterator type*  *a and b: objects of this iterator type*  *t: object of the type pointed by the iterator type*  *n: integer value* | | | | | |

## Constant Iterators

Declaring an iterator const is like declaring a pointer const – the iterator isn’t allowed to point to something different, but the thing it points to may be modified.

Example:

std::vector<int> vec;

...

const std::vector<int>::iterator iter = vec.begin();

// iter acts like a T\* const pointer

\*iter = 10; // OK, changes what iter points to

++iter; // Error, iter is const

std::vector<int>::const\_iterator cIter = vec.begin();

// cIter acts like a const T\* pointer

\*cIter = 10; // Error!

++cIter; // OK

## Benefits of Iterators

Iterators are extremely useful. Some of the benefits of using iterators are as listed below:

**Work for All Containers**

If we use [] operator for a container, then this container has to be in a form of array. Some containers, such as std::list, do NOT offer [] because they do NOT support random-access iterators. But if you use iterators, you can use that code on any std::list implementation. That’s because **most STL containers provide iterators**.

Also, by using iterator your code becomes more portable and generic between containers. You can switch containers from std::vector to std::list or other container freely without changing much if you use iterator. Such rule doesn't apply to operator [].

**Well Connected to Algorithms**

Iterators provide the [glue between algorithms and containers](https://stackoverflow.com/questions/11947151/why-is-there-a-separation-of-algorithms-iterators-and-containers-in-c-stl/11948413#11948413). For generic code, the recommendation would be to use a combination of STL algorithms (*find*, *sort*, *remove*, *copy*, etc. those **return value is an iterator**) that carry out the computation on your data structure (*vector*, *list*, *map*, etc.), and to supply that algorithm with iterators into your container.

Algorithm Library:

<http://www.cplusplus.com/reference/algorithm/>

**Dynamic Processing of Container**

Iterators provide us the ability to **dynamically add or remove elements from the container** with ease. Doing the same without iterators would have been very tedious as it would require shifting the elements every time before insertion and after deletion.

Another advantage of iterators is that it **doesn't assume the data is resident in memory**. For example, one could create a forward iterator that can read data from an input stream, or that simply generates data on the fly (e.g. a range or random number generator).

#include <iostream>

#include <vector>

using namespace std;

int main() {

    // Declaring a vector

    vector<int> v = {1, 2, 3};

    // Declaring an iterator

    vector<int>::iterator it;

    // Inserting element using iterators

    it = v.begin();

    v.insert(it, 5);    // inserting 5 at the beginning of v

    // output: v = {5 1 2 3}

    // Deleting element using iterators

    it = v.begin() + 1;

    v.erase(it);          // it now points to the element after the deleted element

    // output: v = {5 2 3}

    // Accessing the elements using iterators, from the first to the last element

    for (it = v.begin(); it != v.end(); ++it) {

        cout << \*it << " ";

    }

    cout << endl;

    // Accessing the elements using iterators, from the last to the first element

    vector<int>::reverse\_iterator rit;

    for (rit = v.rbegin(); rit != v.rend(); ++rit) {

        cout << \*rit << " ";

    }

    /\*

    Note:

      begin(): The first element.

      end(): The element after the last element. It doesn't exist.

      rbegin(): The last element.

      rend(): The element before the first element. It doesn't exist.

    \*/

    return 0;

}

Output:

5 2 3

3 2 5

# STL

## Algorithm

Header file: <algorithm>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Function** | **Parameters** | **Description** | **Since** | **Example Using Function** | **Traditional Equivalent** |
| std::any\_of | InputIt first,  InputIt last,  UnaryPredicate p | Returns true if any element in the range satisfies the predicate | C++11 | #include <iostream>  #include <algorithm>  #include <vector>  int main() {  std::vector<int> v = {1, 3, 5, 6};  bool result = std::any\_of(  v.begin(),  v.end(),  [](int x) {  return x % 2 == 0;  });  std::cout << std::boolalpha << result; // true } | #include <iostream>  #include <vector>  int main() {  std::vector<int> v = {1, 3, 5, 6};  bool result = false;  for (int x : v) {  if (x % 2 == 0) {  result = true;  break;  }  }  std::cout << std::boolalpha << result;// true } |
| std::all\_of | InputIt first,  InputIt last,  UnaryPredicate p | Returns true if all elements satisfy the predicate | C++11 | #include <iostream>  #include <algorithm>  #include <vector>  int main() {  std::vector<int> v = {2, 4, 6};  bool result = std::all\_of(  v.begin(),  v.end(),  [](int x) {  return x % 2 == 0;  });  std::cout << std::boolalpha << result; // true } | #include <iostream>  #include <vector>  int main() {  std::vector<int> v = {2, 4, 6};  bool result = true;  for (int x : v) {  if (x % 2 != 0) {  result = false;  break;  }  }  std::cout << std::boolalpha << result;// true } |
| std::none\_of | InputIt first,  InputIt last,  UnaryPredicate p | Returns true if no elements satisfy the predicate | C++11 | #include <iostream>  #include <algorithm>  #include <vector>  int main() {  std::vector<int> v = {1, 3, 5};  bool result = std::none\_of(  v.begin(),  v.end(), [](int x) {  return x % 2 == 0;  });  std::cout << std::boolalpha << result; // true } | #include <iostream>  #include <vector>  int main() {  std::vector<int> v = {1, 3, 5};  bool result = true;  for (int x : v) {  if (x % 2 == 0) {  result = false;  break;  }  }  std::cout << std::boolalpha << result;// true } |
| std::find\_if | InputIt first,  InputIt last,  UnaryPredicate p | Finds the first element satisfying the predicate | C++11 | #include <iostream>  #include <algorithm>  #include <vector>  int main() {  std::vector<int> v = {1, 3, 4, 5};  auto it = std::find\_if(  v.begin(),  v.end(),  [](int x) {  return x % 2 == 0;  });  std::cout << \*it; // 4 } | #include <iostream>  #include <vector>  int main() {  std::vector<int> v = {1, 3, 4, 5};  int result = -1;  for (int x : v) {  if (x % 2 == 0) {  result = x;  break;  }  }  std::cout << result; // 4 } |
| std::copy\_if | InputIt first,  InputIt last,  OutputIt d\_first,  UnaryPredicate p | Copies elements that satisfy the predicate | C++11 | #include <iostream>  #include <algorithm>  #include <vector>  int main() {  std::vector<int> v = {1, 2, 3, 4};  std::vector<int> evens;  std::copy\_if(  v.begin(),  v.end(),  std::back\_inserter(evens),  [](int x) {  return x % 2 == 0;  });  for (int x : evens) {  std::cout << x << ' '; // 2 4  }  } | #include <iostream>  #include <vector>  int main() {  std::vector<int> v = {1, 2, 3, 4};  std::vector<int> evens;  for (int x : v) {  if (x % 2 == 0) {  evens.push\_back(x);  }  }  for (int x : evens) {  std::cout << x << ' '; // 2 4  } } |
| std::count\_if | InputIt first, InputIt last, UnaryPredicate p | Counts elements satisfying a predicate | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  int count = std::count\_if(  v.begin(),  v.end(),  [](int x) {  return x % 2 == 0;  });  std::cout << count; // 2  } | #include <iostream>  #include <vector>    int main    () {  std::vector<int> v = {1, 2, 3, 4, 5};  int count = 0;  for (int x : v) {  if (x % 2 == 0) ++count;  }  std::cout << count; // 2  } |
| std::remove\_if | ForwardIt first,  ForwardIt last,  UnaryPredicate p | Removes elements satisfying a predicate (returns new end iterator) | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  auto new\_end = std::remove\_if(  v.begin(),  v.end(),  [](int x) {  return x % 2 == 0;  });  v.erase(new\_end, v.end());  for (int x : v) {  std::cout << x << ' '; // 1 3 5  }  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  std::vector<int> result;  for (int x : v) {  if (x % 2 != 0) result.push\_back(x);  }  for (int x : v) {  std::cout << x << ' '; // 1 3 5  }  } |
| std::transform | InputIt first1,  InputIt last1, OutputIt d\_first,  UnaryOperation op | Applies a function to each element and stores the result | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3};  std::vector<int> squared(v.size());  std::transform(  v.begin(),  v.end(),  squared.begin(),  [](int x) {  return x \* x;  });  for (int x : squared) {  std::cout << x << ' '; // 1 4 9  }  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3};  std::vector<int> squared;  for (int x : v) {  squared.push\_back(x \* x);  }  for (int x : squared) {  std::cout << x << ' '; // 1 4 9  }  } |
| std::equal | InputIt1 first1,  InputIt1 last1,  InputIt2 first2 | Checks if two ranges are equal | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> a = {1, 2, 3};  std::vector<int> b = {1, 2, 3};  bool result = std::equal(a.begin(), a.end(), b.begin());  std::cout << std::boolalpha << result; // true  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> a = {1, 2, 3};  std::vector<int> b = {1, 2, 3};  bool result = true;  for (size\_t i = 0; i < a.size(); ++i) {  if (a[i] != b[i]) {  result = false;  break;  }  }  std::cout << std::boolalpha << result; // true  } |
| std::is\_sorted | ForwardIt first,  ForwardIt last | Checks if the range is sorted | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4};  bool result = std::is\_sorted(v.begin(), v.end());  std::cout << std::boolalpha << result; // true  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4};  bool result = true;  for (size\_t i = 1; i < v.size(); ++i) {  if (v[i] < v[i - 1]) {  result = false;  break;  }  }  std::cout << std::boolalpha << result; // true  } |
| std::partition | ForwardIt first,  ForwardIt last,  UnaryPredicate p | Reorders elements so that those satisfying the predicate come before others | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  std::partition(v.begin(), v.end(), [](int x){ return x % 2 == 0; });  for (int x : v) {  std::cout << x << ' '; // 2 4 1 3 5  }  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  std::vector<int> evens, odds;  for (int x : v) {  if (x % 2 == 0) evens.push\_back(x);  else odds.push\_back(x);  }  v.clear();  v.insert(v.end(), evens.begin(), evens.end());  v.insert(v.end(), odds.begin(), odds.end());  for (int x : v) {  std::cout << x << ' '; // 2 4 1 3 5  }  } |
| std::unique | ForwardIt first,  ForwardIt last | Removes consecutive duplicates (returns new end iterator) | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v = {1, 1, 2, 2, 3, 3};  auto new\_end = std::unique(v.begin(), v.end());  v.erase(new\_end, v.end());  for (int x : v) {  std::cout << x << ' '; // 1 2 3  }  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {1, 1, 2, 2, 3, 3};  std::vector<int> result;  for (size\_t i = 0; i < v.size(); ++i) {  if (i == 0 || v[i] != v[i - 1]) {  result.push\_back(v[i]);  }  }  for (int x : v) {  std::cout << x << ' '; // 1 2 3  }  } |
| std::reverse | BidirectionalIt first,  BidirectionalIt last | Reverses the order of elements in a range | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3};  std::reverse(v.begin(), v.end());  for (int x : v) {  std::cout << x << ' '; // 3 2 1  }  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3};  for (int i = v.size()-1; i >= 0; --i) {  std::cout << v[i] << ' ';  }  } |
| std::rotate | ForwardIt first,  ForwardIt middle,  ForwardIt last | Rotates elements in the range so that middle becomes the first element | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  std::rotate(v.begin(), v.begin() + 2, v.end());  for (int x : v) {  std::cout << x << ' '; // 3 4 5 1 2  }  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  std::vector<int> rotated;  rotated.insert(rotated.end(), v.begin() + 2, v.end());  rotated.insert(rotated.end(), v.begin(), v.begin() + 2);  for (int x : v) {  std::cout << x << ' '; // 3 4 5 1 2  }  } |
| std::fill | ForwardIt first,  ForwardIt last,  const T& value | Assigns the given value to the elements in the range | C++11 | #include <iostream>  #include <algorithm>  #include <vector>    int main() {  std::vector<int> v(5);  std::fill(v.begin(), v.end(), 7);  for (int x : v) {  std::cout << x << ' '; // 7 7 7 7 7  }  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v(5);  for (size\_t i = 0; i < v.size(); ++i) {  v[i] = 7;  }  for (int x : v) {  std::cout << x << ' '; // 7 7 7 7 7  }  } |

## Iterator

Header file: <iterator>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Function** | **Parameters** | **Description** | **Since** | **Example Using Function** | **Traditional Equivalent** |
| std::next | InputIt it,  Distance n = 1 | Returns an iterator advanced by n positions | C++11 | #include <iostream>  #include <iterator>  #include <vector>    int main() {  std::vector<int> v = {10, 20, 30, 40};  auto it = std::next(v.begin(), 2);  std::cout << \*it; // 30  } |  |
| std::prev | BidirectionalIt it,  Distance n = 1 | Returns an iterator moved backward by n positions | C++11 | #include <iostream>  #include <iterator>  #include <vector>    int main() {  std::vector<int> v = {10, 20, 30, 40};  auto it = std::prev(v.end(), 2);  std::cout << \*it; // 30  } |  |
| std::advance | InputIt& it,  Distance n | Advances the iterator by n steps | C++11 | #include <iostream>  #include <iterator>  #include <vector>    int main() {  std::vector<int> v = {5, 10, 15, 20};  auto it = v.begin();  std::advance(it, 3);  std::cout << \*it; // 20  }  #include <iostream>  #include <iterator>  #include <vector>    int main() {  std::vector<int> v = {10, 20, 30, 40};  auto it = v.end();  std::advance(it, -2);  std::cout << \*it; // 30  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {5, 10, 15, 20};  auto it = v.begin();  for (int i = 0; i < 3; ++i) ++it;  std::cout << \*it; // 20  } |
| std::distance | InputIt first,  InputIt last | Returns the number of steps between two iterators | C++11 | #include <iostream>  #include <iterator>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  std::cout << std::distance(v.begin(), v.end()) ; // 5  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> v = {1, 2, 3, 4, 5};  int count = 0;  for (auto it = v.begin(); it != v.end(); ++it) ++count;  std::cout << count; // 5  } |
| std::back\_inserter | Container& c | Returns a back insert iterator that appends to the container | C++11 | #include <iostream>  #include <iterator>  #include <vector>  #include <algorithm>    int main() {  std::vector<int> src = {1, 2, 3};  std::vector<int> dest;  std::copy(src.begin(), src.end(), std::back\_inserter(dest));  for (int x : dest) std::cout << x << ' '; // 1 2 3  } | #include <iostream>  #include <vector>    int main() {  std::vector<int> src = {1, 2, 3};  std::vector<int> dest;  for (int x : src) dest.push\_back(x);  for (int x : dest) std::cout << x << ' '; // 1 2 3  } |

## Type

Std::boolalpha

# Function Overloading

## Definition

Function overloading allows **multiple definitions of a function by changing its signature**, including **number of parameters** and **datatype of parameters**. Return type plays NO role.

Notes:

* Function overloading is achieved at **compile time**.
* It can be done in **base and derived class**.
* Overloaded functions are in **same scope**.
* **C doesn’t support function overload** (because C doesn’t support static polymorphism)

When you call an overloaded function, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

## Example

Following is the example where same function **print()** is being used to print different data types:

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print float

pd.print(500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

Output:

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

# Operator Overloading

**Note**: C doesn’t support operator overload (because C doesn’t support static polymorphism).

## Definition

You can **redefine most of the built-in operators** available in C++. Thus, a programmer can use operators with user-defined types.

Overloaded operators are created using keyword operator followed by the symbol for the operator being defined (+, -, etc.).Like any other function, **an overloaded operator has a return type and a parameter list**.

## Creating

Let’s declares an *addition operator* that can be used to **add** two Box objects and returns final Box object.

Way 1: as a class member function

Box operator+(const Box&)

Way 2: as a class non-member function

Box operator+(const Box&, const Box&);

## Example

Example 1: as a class member function

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void) {

return length \* width \* height;

}

void setLength( double len ) {

length = len;

}

void setWidth( double wid ) {

width = wid;

}

void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.width = this->width + b.width;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double width; // Width of a box

double height; // Height of a box

};

// Main function for the program

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setWidth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setWidth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

Output:

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

## Overloadable/Non-Overloadable Operators

Following is the list of operators which can be overloaded:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | ^ |
| & | | | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | || |
| += | -= | /= | %= | ^= | &= |
| |= | \*= | <<= | >>= | [] | () |
| -> | ->\* | new | new [] | delete | delete [] |

Following is the list of operators, which CANNOT be overloaded:

|  |  |  |  |
| --- | --- | --- | --- |
| :: | .\* | . | ?: |

## Types of Operator Overloading

There are various operator overloading types:

|  |  |
| --- | --- |
| **No** | **Operators & Example** |
| 1 | [Unary Operators Overloading](https://www.tutorialspoint.com/cplusplus/unary_operators_overloading.htm) |
| 2 | [Binary Operators Overloading](https://www.tutorialspoint.com/cplusplus/binary_operators_overloading.htm) |
| 3 | [Relational Operators Overloading](https://www.tutorialspoint.com/cplusplus/relational_operators_overloading.htm) |
| 4 | [Input/Output Operators Overloading](https://www.tutorialspoint.com/cplusplus/input_output_operators_overloading.htm) |
| 5 | [++ and -- Operators Overloading](https://www.tutorialspoint.com/cplusplus/increment_decrement_operators_overloading.htm) |
| 6 | [Assignment Operators Overloading](https://www.tutorialspoint.com/cplusplus/assignment_operators_overloading.htm) |
| 7 | [Function call () Operator Overloading](https://www.tutorialspoint.com/cplusplus/function_call_operator_overloading.htm) |
| 8 | [Subscripting [] Operator Overloading](https://www.tutorialspoint.com/cplusplus/subscripting_operator_overloading.htm) |
| 9 | [Class Member Access Operator -> Overloading](https://www.tutorialspoint.com/cplusplus/class_member_access_operator_overloading.htm) |

## Why Operator Overloading?

* Readability: allow to use familiar operators to perform operations on user-defined types (especially, **mathematic or logic expressions, string modulation**, etc.). For example, you can write vector1 + vector2 instead of invoking a separate method like vector1.add(vector2).

# Copy Initialization and Direct Initialization

In C++, there are 3 ways to initialize a variable:

int value1 = 1; // copy initialization

double value2(2.2); // direct initialization

char value3{'c'}; // uniform initialization (only C++11)

**Copy Initialization**:

* Form: T x = a.
* Used in argument passing, function return, throwing/handling an exception, and brace-enclosed initializer lists.
* In class data type, it’s *copy constructor* which initializes object using another object of the same class.

**Direct Initialization**:

* Form: T x(a).
* Used in new expressions, static\_cast expressions, functional notation type conversions, and base and member initializers.
* In class data type, it’s *default constructor with parameters* being called and this takes place as soon as a class object is created.

**Uniform Initialization**:

* Equivalent to *direct initialization*, but it’s only available in C++11.

**Example**:

Foo\* f = new Foo(2,3); // "=" is copy init, but the RHS is direct init

int x = static\_cast<int>(d); // "=" is copy init, but the RHS is direct init

try {

throw "error"; // copy init ("error" is the copy of string object)

} catch(string s) { } // copy init ("s" is passed by value)

int foo(int &p, int q) { // copy init ("q" is passed by value)

return p+q; // copy init (temporary holder for return)

}

const int& foo(int& p, const int& q) { // direct init ("q" is passed by ref)

int res(p+q); // direct init

return res; // direct init (no temp holder, but ref)

}

**Why Direct Initializations, NOT Copy Initializations?**

For variables of primitive data types, **copy** and **direct** initializations are generally similar. However, for objects of class types, they differ quite significantly in sematic.

With direct initialization, the constructor is called directly to construct the object. In contrast, **copy initialization involves creating a temporary object** and then copying it into the target object. By using direct initialization, you eliminate the overhead of creating and copying temporary objects.

**Rule of Thumb**:

Foo b(a); // should

Foo b{a}; // should (C++11 only)

Foo c = a; // avoid

# Uniform Initialization (C++11)

C++ has at least 4 different initialization notations:

Parenthesized initialization looks like this:

std::string s("hello");

int m = int();

You can use the = notation for the same purpose in certain cases:

std::string s = "hello";

int x = 5;

For Plain Old Data (POD) aggregates, you use braces:

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

struct tm today = {0};

Finally, constructors use member initializers:

struct S {

int x;

S(): x(0) {}

};

This is confusing! Worse yet, in C++03 you can't initialize POD array members and POD arrays allocated using new[].

**C++11 cleans up this mess with a uniform brace notation** {}:

std::string s{"hello"};

class Foo {

public:

Foo(int i, int j);

};

Foo foo{0, 0}; // C++11 only. Equivalent to: Foo foo(0, 0);

int\* arr = new int[4] {0, 1, 2, 3}; // C++11 only

struct Bar {

int arr[4];

Bar() : arr{1, 2, 3, 4} {} // C++11 only

};

Even better, you can say goodbye to a long list of push\_back() calls:

std::vector<string> vec = {"first", "second", "third"};

std::map singers = { {"Lady Gaga", "+1(212)555-7890"},

{"Beyonce Knowles", "+1(212)555-0987"} };

# Class and Object

The main purpose of C++ is to add *Object Orientation* to C, and **classes are the central feature of C++**.

## Class Definition

A class is a ***user-defined data type* that serves as a blueprint for creating objects**. It encapsulates *data* (member variables) and *methods* (member functions) that operate on that data.

In C++, a class definition starts with keyword class followed by the class name and the class body (enclosed by a pair of {} and followed by a ;).

For example, we defined the Box data type as follows:

class Box {

private:

double length; // Length of a box

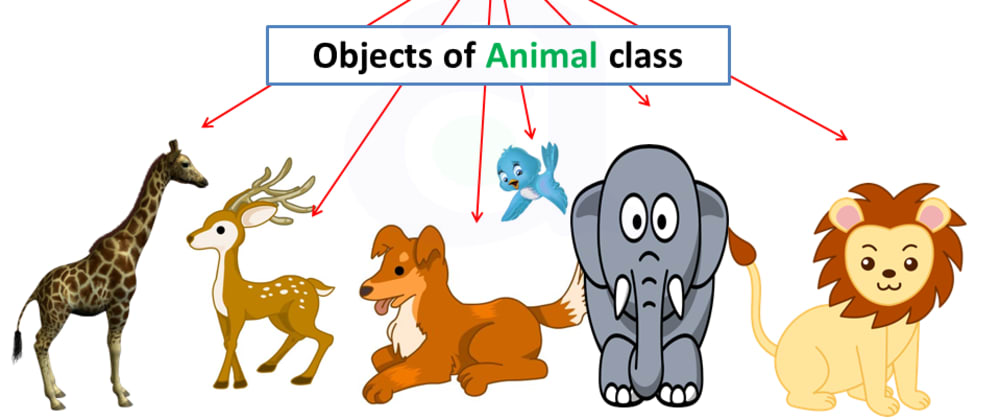
double width; // Width of a box

double height; // Height of a box

};

## Class Instance (Object)

An **object is an instance of a class** with specific value and state.



We declare object with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will **have their own copy of data members**.

## Accessing Members

To access data members of a class, we use a member access operator (.).

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double width; // Width of a box

double height; // Height of a box

};

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.width = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.width = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.width;

cout << "Volume of Box1: " << volume <<endl;

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.width;

cout << "Volume of Box2: " << volume <<endl;

return 0;

}

Output:

Volume of Box1: 210

Volume of Box2: 1560

## Classes and Objects in Detail

|  |  |
| --- | --- |
| **Concept** | **Description** |
| [Class Member Functions](https://www.tutorialspoint.com/cplusplus/cpp_class_member_functions.htm) | A member function of a class (or *method*) is a function that has its definition or its prototype within the class definition like any other variable. |
| [Class Access Modifiers](#_Access_Specifiers) | A class member can be defined as *public*, *private* or *protected*. By default, members would be assumed as private. |
| [Constructor & Destructor](https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm) | A class constructor is a special function in a class that is called first when a new object of the class is created. A destructor is also a special function which is called when created object is deleted. |
| Copy Constructor | A copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. |
| Copy Assignment Operator | A copy assignment operator is used to assign one object to another of the same type. |
| Scope Resolution | A scope resolution operator is denoted by a :: symbol. It resolves the barrier of scope in a program. A scope resolution operator is used to reference a member out of their scope. |
| Friend Class and Friend Function | A *friend class* can access private, protected, and public members of other classes in which it is declared as friends.  A *friend function* can also access private, protected, and public members. But, it’s not a member function of the class. |
| [The this Pointer](https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm) | Every object has a special pointer this which points to the object itself. |
| [Pointer to C++ Classes](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_class.htm) | A pointer to a class is done exactly the same way a pointer to a structure is. In fact, a class is really just a structure with functions in it. |
| Static Members of Classes | Both data members and function members of a class can be declared as static. |
| Const Member Function | Values of objects cannot be modified within this function. |
| Initializer List |  |
| Conversion Operator | A conversion operator is a special kind of member function that converts a value of a class type to a value of some other type. |
| Forward Declaration |  |

### Default Constructor

* A constructor which can be invoked without any arguments. That doesn’t mean that it can’t take any parameters; but if it can take parameters, it has to provide default values for them.
* C++ compiler implicitly provides a default constructor, if no default constructor is defined in the class.

### Copy Constructor

A copy constructor is a constructor which **creates an object by initializing it with an object of the same class**, which has been created previously.

The most common form of copy constructor is:

classname(const classname &obj) {

// body of constructor

}

It’s used to:

* **Initialize an object from another of the same type**: The new object has the same values as the original one, but can be operated independently.
* **Pass a copy of object as an argument to a function**: Pass a new object to the function without affecting the original one.
* **Return a copy of object from a function**: Return a new object to the caller without affecting the original one.
* **Deeply copy the object**: [Shadow copy](#_Shadow_Copy) only copies the memory address stored in pointer, leading to two objects sharing the same memory. If your class has pointer members, you should copy the class object using deep copy. And to do deep copy, a custom copy constructor is required.

**Notes**:

* If a copy constructor is not defined, the compiler itself defines one. But it’s **recommended to define our own copy constructor if objects consist of pointers or dynmic memory allocation**.

**Example 1**: Initialize one object from another of the same type

#include <iostream>

using namespace std;

class Line {

private:

    int \*ptr;

public:

    Line(int len);              // param constructor

    Line(const Line &line);     // copy constructor

    ~Line();                    // destructor

    int getLength();

};

Line::Line(int len) {

    cout << "Param constructor" << endl;

    ptr = new int;

    \*ptr = len;

}

Line::Line(const Line &line) {

    cout << "Copy constructor" << endl;

    ptr = new int; // deep copy

    \*ptr = \*line.ptr;

}

Line::~Line() {

   cout << "Destructor" << endl;

   delete ptr;

}

int Line::getLength() {

   return \*ptr;

}

int main() {

   Line line(10); // this calls param constructor

   Line line2 = line;   // this calls copy constructor

   return 0;

}

Output:

Param constructor

Copy constructor

Destructor

Destructor

**Example 2**: Copy an object to pass it as an argument to a function

Based on example 1, but add a new function and change the main function a little bit:

...

void display(Line line) {

   cout << "Length of line: " << line.getLength() <<endl;

}

int main() {

   Line line(10); // this calls param constructor

   display(line); // this calls copy constructor

   return 0;

}

Output:

Param constructor

Copy constructor

Length of line: 10

Destructor

Destructor

**Example 3**: Copy an object to return it from a function

…

### Copy Assignment Operator

Copy assignment operator is used to **assign one object to another of the same type**.

* It is called when an already initialized object is assigned a new value from another existing object.
* It does not create a separate memory block or new memory space.

It has a general form:

MyClass& operator=(const MyClass& other) {

...

return \*this;

}

**Example 1**:

#include <iostream>

class Number {

public:

    int value;

    Number(int val) : value(val) {}

    Number& operator=(const Number& other) {

        if (this != &other) {

            value = other.value;

        }

        return \*this;

    }

};

int main() {

    Number obj1(1);

    Number obj2(2);

    obj2 = obj1;    // Copy assignment

    std::cout << obj2.value << std::endl;  // Output: 2

    return 0;

}

**Example 2**: A quick comparison of default constructor, copy construct and copy assignment operator.

#include <iostream>

using namespace std;

class Foo {

public:

   Foo() {

       cout << "default constructor" << endl;

   }

   Foo(const Foo&) {

       cout << "copy constructor" << endl;

   }

   const Foo& operator=(const Foo&) {

       cout << "copy assignment operator" << endl;

       return \*this;

   }

};

int main() {

   Foo a;       // call default constructor -> direct initialization

   Foo b(a);    // call copy constructor -> direct initialization

   Foo c{a};    // call copy constructor -> uniform initialization

   Foo d = a;   // call copy constructor -> copy initialization

   c = b;       // call copy assignment operator

   return 0;

}

Output:

default constructor

copy constructor

copy constructor

copy constructor

copy assignment operator

### Friend Class and Friend Function

A *friend class* can access private, protected, and public members of other classes in which it is declared as friends.

A *friend function* can also access private, protected, and public members. But, it’s not a member function of the class.

**Example 1**: Friend class

#include <iostream>

class Box {

private:

    double width;

public:

    Box() : width(0.0) {}

    void setWidth(double w) {

        width = w;

    }

    friend class Output;  // A friend class

};

class Output {

public:

    // Because this class is friend of class Box, it can access private members "width" of Box

    void printWidth(const Box& box) {

        std::cout << "Width of Box: " << box.width << std::endl;

    }

};

int main() {

    Box box;

    Output output;

    box.setWidth(3.5);

    output.printWidth(box);

    return 0;

}

**Example 2**: Friend function

#include <iostream>

using namespace std;

class Box {

private:

   double width;

public:

    friend void printWidth(Box box);    // Friend function

    void setWidth(double w) {

        width = w;

    }

};

void printWidth(Box box) {

   // Because this is a friend of Box, it can access any member of this class

   cout << "Width of box: " << box.width <<endl;

}

int main() {

   Box box;

   box.setWidth(10.0);

   printWidth(box);

   return 0;

}

### Static Members of Classes

#### Static Variables

Variables declared as static are **shared by all objects of the class**. That means they can be called even if no objects of the class exist. They have a lifetime that extends throughout the entire program.

There cannot be multiple copies of the same static variable for different objects. So we cannot initialized it in the class definition (including constructor), but it can **only be initialized once OUTSIDE the class** using the scope resolution operator (::).

For example, the below program fails to compile:

#include<iostream>

using namespace std;

class A {

public:

static int i;

A()

{

// Do nothing

};

};

int main()

{

A obj;

obj.i = 2; // Cause error: undefined reference to 'A::i'

}

But this program works well:

#include<iostream>

using namespace std;

class A

{

public:

static int i;

A()

{

// Do nothing

};

};

// static variable must be initialized explicitly outside of the class

int A::i = 1;

int main()

{

A obj;

cout << obj.i;

}

**Notes**:

* Static member variables do not have access to the this pointer. So, they can **only be used in static member functions**.

#### Static Functions

Just like class static variables, static functions **don’t depend on any object of class**. They have global scope.

We call them using the object and the '.' operator, but it is recommended to do that with the '::' operator.

**Example**:

#include<iostream>

using namespace std;

class A {

public:

// static member function

static void printMsg()

{

cout << "Welcome to A!";

}

};

// main function

int main()

{

// call a static member function

A::printMsg();

}

**Notes**:

* Static member functions do not have access to the this pointer. So, they can **only access static member variables or other static member functions**.

However, if you want to access non-static members inside a static member function, you can pass the object instance explicitly to the static function. Here's an example:

#include<iostream>

using namespace std;

class MyClass {

public:

int nonStaticMember;

public:

static void StaticMethod(MyClass& instance) {

// Access non-static member through the passed instance

instance.nonStaticMember = 1;

}

};

int main() {

MyClass myClass;

MyClass::StaticMethod(myClass);

cout << myClass.nonStaticMember << endl;

return 0;

}

* Static member functions cannot be declared as const or volatile, as these qualifiers apply to instance-specific behavior.
* Static member functions are often used for **utility functions** or operations that are not tied to any specific instance of the class. They can provide common functionality, or act as factory methods to create class instances.
* If a static member function in a derived class has the same name as a non-static member function in the base class, the compiler will hide the non-static one. This can lead to unexpected behavior.

### Constant Member Functions

A function becomes constant when the const keyword is used in function’s declaration. The idea of constant functions is to **prevent modification of objects (mostly member variables) called in the function**.

It is recommended practice to make as many functions constant as possible so that accidental changes to objects are avoided.

For example:

#include<iostream>

using namespace std;

class Test {

int value;

public:

// We'll get COMPILER ERROR if we add a line like "value = 100;" in this function.

int getValue() const {

return value;

}

};

**Notes**:

* Constant function can be called on **any type of object**. In other words, **non-const functions can only be called by non-const objects**. For example, the following program has COMPILER ERRORS.

#include<iostream>

using namespace std;

class Test {

int value;

public:

int getValue() { return value; }

};

int main() {

const Test t;

// COMPILE ERROR: passing 'const Test' as 'this' argument of 'int

cout << t.getValue();

return 0;

}

### Initializer List

**Reading uninitialized values yields undefined behavior**. On some platforms, the mere act of reading an uninitialized value can halt your program. More typically, **the result of the read will be semi-random bits**, which will then pollute the object you read the bits into, eventually leading to inscrutable program behavior and a lot of unpleasant debugging.

So the rule is to always initialize your objects before you use them.

**For non-member objects of built-in types:**

You can use *copy initialization* as follows:

int x = 0;

const char\* text = "";

double d;

std::cin >> d;

**For non-member objects of class types:**

Instead of copy initialization, you should use *direct initialization* as follows:

std::string s("Hello World"); // should

std::string s = "Hello World"); // avoid

**For member objects:**

The responsibility for initialization falls on constructors:

class PhoneNumber { ... };

// Address Book Entry

class ABEntry {

public:

ABEntry(const std::string& name,

const std::string& address,

const std::list<PhoneNumber>& phones);

private:

std::string mName;

std::string mAddress;

std::list<PhoneNumber> mPhones;

int mNumTimesConsulted;

};

ABEntry::ABEntry(const std::string& name,

const std::string& address,

const std::list<PhoneNumber>& phones)

{

mName = name; // AVOID!

mAddress = address; // These are all assignments,

mPhones = phones; // NOT initializations

mNumTimesConsulted = 0;

}

This yields objects with the values you expect, but it’s not the best approach. That’s because inside the ABEntry constructor, mName, mAddress, and mPhones aren’t being initialized, they’re being *assigned*. This is similar to:

{

std::string mName; // call defualt ctor

std::string mAddress;

std::list<PhoneNumber> mPhones;

int mNumTimesConsulted;

mName = name; // call operator =

mAddress = address;

mPhones = phones;

mNumTimesConsulted = 0;

}

The rules of C++ stipulate that **data members of an object should be *initialized* before the body of a constructor is entered**. In other words, initialization took place earlier — when their default constructors were automatically called prior to entering the body of the ABEntry constructor.

A better way to write the ABEntry constructor is using the ***Member Initialization List***:

ABEntry::ABEntry(const std::string& name,

const std::string& address,

const std::list<PhoneNumber>& phones)

: mName(name), // These are now all initializations

mAddress(address),

mPhones(phones),

mNumTimesConsulted(0)

{} // the ctor body is now empty

Using a Member Initialization List is almost identical to doing [direct initialization](#_gjdgxs) (or uniform initialization in C++11).

This constructor yields the same output as the one above, but it will be more efficient. The old assignment-based version first called *default constructors* to initialize mName, mAddress, and mPhones, then assigned them with new values on top of default-constructed ones. All the work performed in those default constructions was wasted.

However, the member initialization list avoids that problem, because the arguments in the list are used as constructor arguments for data members. In this case, mName is copy-constructed from name, mAddress is copy-constructed from address, and mPhones is copy-constructed from phones.

For most types, **a single call to a copy constructor is more efficient — sometimes *much* more efficient — than a call to the default constructor followed by a call to the copy assignment operator**.

For objects of built-in type like mNumTimesConsulted, there is no difference in cost between initialization and assignment, but for consistency, it’s often best to initialize everything via member initialization list.

Similarly, you can use the member initialization list even when you want to default-construct a data member; just specify nothing as an initialization argument. For example, if ABEntry had a constructor taking no parameters, it could be implemented like this:

ABEntry::ABEntry()

: mName(), // call mName’s default ctor

mAddress(), // do the same for mAddress and mPhones

mPhones(),

mNumTimesConsulted(0) // but explicitly initialize numTimesConsulted to zero

{}

Sometimes the initialization list must be used, even for built-in types. For example, **data members that are const or are references must be initialized; they can’t be assigned**. To avoid having to memorize when data members must be initialized in the member initialization list and when it’s optional, the easiest choice is to always use the initialization list. It’s sometimes required, and it’s often more efficient than assignments.

### Conversion Operators

A conversion operator is a special kind of member function that **converts a value of a class type to a value of some other type**.

Its general form is:

operator *type*() const;

Where:

* type: a data type which will be converted to from the class type.

For example:

#include <cmath>

#include <iostream>

using namespace std;

class Complex {

private:

    double real;

    double imag;

public:

    Complex(double r = 0.0, double i = 0.0)

        : real(r), imag(i)

    {

    }

    double mag() {

        return getMag();

    }

    // Allow an object of type "Complex" to be implicitly converted to type "double"

    operator double() {

        return getMag();

    }

private:

    double getMag()

    {

        return sqrt(real \* real + imag \* imag);

    }

};

int main()

{

    Complex complex(3.0, 4.0);

    // 1st way to print magnitude

    cout << complex.mag() << endl;

    // 2nd way to print magnitude (thanks to conversion operator)

    cout << complex << endl;

}

Output:

5

5

**Note**: Conversion operators are NOT easy to use. They can lead to ambiguity. They can also cause unexpected behaviors.

### Using 'explicit' Keyword

The explicit keyword is is used to **mark conversion constructors** **and conversion operators to NOT implicitly convert types** in C++. If a class has a constructor which can be called with a single argument, then this constructor becomes a *conversion constructor* because it allows conversion of the single argument to the class being constructed.

But such a conversion can cause problems. The explicit keyword is created to help prevent unintended and potentially unsafe conversions.

**Example 1**: Conversion constructor

#include <iostream>

using namespace std;

class MyNumber {

private:

    int num;

public:

    explicit MyNumber(int n) : num(n) {}

    int getValue() const {

        return num;

    }

};

int main() {

    MyNumber num(10);

    // Due to explicit constructor, the below code causes ERROR:

    // conversion from ‘int’ to non-scalar type ‘MyNumber’ requested

    // MyNumber num = 10;

    cout << num.getValue() << endl;

    return 0;

}

**Example 2**: Conversion operator

Let’s reuse the example in the [Conversion Operator session](#_Conversion_Operator), but this time, we disallow the implicit conversion:

class Complex {

...

    // Allow an object of type "Complex" to be converted to type "double"

    explicit operator double() {

        return getMag();

    }

int main()

{

    ...

    // 2nd way to print magnitude (thanks to conversion operator)

    cout << static\_cast<double>(complex) << endl;

// The below code causes ERROR:

// no match for ‘operator<<’ (operand types are ‘std::ostream’ {aka ‘std::basic\_ostream’} and ‘Complex’)

// cout << complex << endl;

}

As seen, the type conversion is now explicit with static\_cast. Any attempt to implicitly convert Complex to double will cause compilation error.

### Forward Declaration

A forward declaration is a statement that **tells the compiler that a type with a specific name will be defined later**.

If the class or struct is defined in the same file but appears later in the code, or if the two header files include each other, you can use a forward declaration to inform the compiler about the existence of the type.

For example, the code below will give "*error: 'A' does not name a type; did you mean …*" because file a.h include b.h and vice versa.

// In a.h

#pragma once

#include "b.h"

class A {

…

};

// In b.h

#pragma once

#include "a.h"

class B {

A a; // Error here

};

This error can be resolved by using forward declaration:

// In a.h

#pragma once

#include "b.h"

class A {

…

};

// In b.h

#pragma once

// #include "a.h" // --> Don’t do this

class A; // --> Do this (forward declaration)

class B {

// A a; // But this gives error "error: field 'a' has incomplete type 'A'"

A\* a; // To solve it, do this instead

};

However, note that forward declaration is only about the type itself, not objects of that type.

For example, this works:

class B;    // forward declaration

class A {

public:

    B\* b;

    B\* getB() { }

};

class B {

public:

    A a;

    void doB() { }

};

int main() {

    return 0;

}

But this DOES NOT work:

class B;    // forward declaration

class A {

public:

    B\* b;

    void doA() {

        b = new B();    // error: invalid use of incomplete type ‘class B’

        b->doB();       // error: invalid use of incomplete type ‘class B’

    }

};

...

And this DOES NOT work as well:

class B;    // forward declaration

class A {

public:

    B b; // error: invalid use of incomplete type ‘class B’

    void doA() { }

};

...

# OOP Properties

## Inheritance

Inheritance allows us to **define a class in terms of another class**, which makes it easier to create and maintain an application. This also provides an opportunity to **reuse the code** functionality and **improve implementation time**.

When creating a class, instead of writing new code, the programmer can make the new class inherit members of the existing class. This existing class is called the **base class**, and the new class is called the **derived** **class**.

The idea of inheritance is a IS-A relationship. For example, mammal IS-A animal and dog IS-A mammal, so dog IS-A animal too.

### Base Classes and Derived Classes

A class derivation has the form:

class derived-class : access-specifier base-class

Where:

* *access-specifier* is one of public**,** protected**,** or private. If the *access-specifier* is not used, then it is private by default.
* *base-class* is the name of a previously defined class.

Consider a base class Shape and its derived class Rectangle as follows:

#include <iostream>

// Base class

class Shape {

    protected:

        int width;

        int height;

    public:

        void setWidth(int w) {

            width = w;

        }

        void setHeight(int h) {

            height = h;

        }

};

// Derived class

class Rectangle : public Shape {

    public:

        int getArea() {

            return (width \* height);

        }

};

int main() {

    Rectangle Rect;

    Rect.setWidth(5);

    Rect.setHeight(7);

    // Print the area of the object.

    std::cout << "Total area: " << Rect.getArea() << std::endl;

    return 0;

}

Output:

Total area: 35

### Access Specifiers

#### For Class Members

A derived class **can access all the non-private members** (public and protected, but not private) of its base class.

We can summarize the different access types according to who can access them in the following way:

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | **public** | **protected** | **private** |
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

**Note**: A derived class CANNOT inherit following elements from a base class:

* Constructors, destructors and copy constructors of the base class (by default, but we can explicitly inherit).
* Overloaded operators of the base class (by default, but we can explicitly inherit).
* The friend functions of the base class (because friend function is not a class member).

#### For Class Itself

The type of inheritance is specified by the *access-specifier* as explained above.

We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

* **Public** Inheritance − When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class.
* **Protected** Inheritance − When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.
* **Private** Inheritance − When deriving from a private base class, public and protected members of the base class become private members of the derived class.

### Types of Inheritance

#### Single Inheritance



It’s the above "Shape – Rectangle" example.

#### Hierarchical Inheritance



It’s the "Shape – Rectangle – Triangle" example in session "Polymorphism".

#### Multilevel Inheritance



Example:

class Vehicle

{

protected:

    int m\_id;

public:

    Vehicle(int id = 0)

{

}

    ~Vehicle()

    {

    }

};

class Truck : public Vehicle // "Truck" inherits "Vehicle"

{

private:

    int m\_maxWeigtht;

public:

    Truck(int id = 0)

        : Vehicle(id)       // "Truck" inherits constructor of Vehicle

    {

    }

    ~Truck()

    {

    }

};

class Car : public Vehicle // "Car" inherits "Vehicle"

{

private:

    int m\_seatNum;

public:

    Car(int id = 0, int seatNum)

        :  Vehicle(id),         // "Car" inherits constructor of Vehicle

           m\_seatnum(seatNum)   // with an extra attribute

    {

    }

    ~Car()

    {

    }

};

class Roadster : public Car // "Roadster" inherits "Car"

{

private:

    int m\_powerKw;

public:

    Roadster(int id = 0, int seatNum = 0, int m\_powerKw = 0)

        : Car(id, seatNum),   // "Roadster" inherits constructor of Car (and Vehicle also)

m\_powerKw(powerKw)

    {

    }

    ~Roadster()

    {

    }

};

class Camper : public Car // "Camper" inherits "Car"

{

private:

    int m\_sleepingPlaceNum;

public:

    Camper(int id = 0, int seatNum = 0, int sleepingPlaceNum = 0)

        : Car(id, seatNum),   // "Camper" inherits constructor of Car (and Vehicle also)

m\_sleepingPlaceNum(sleepingPlaceNum)

    {

    }

    ~Camper()

    {

    }

};

#### Multiple Inheritance



##### What

**A class can be derived from more than one classes**, which means it can inherit data and functions from multiple base classes:

class derived-class : access-specifier base-classA, access-specifier base-classB ....

Let us try the following example:

#include <iostream>

// Base class Shape

class Shape {

    public:

        void setWidth(int w) {

            width = w;

        }

        void setHeight(int h) {

            height = h;

        }

    protected:

        int width;

        int height;

};

// Base class PaintCost

class PaintCost {

    public:

        int getCost(int area) {

            return area \* 70;

        }

};

// Derived class

class Rectangle : public Shape, public PaintCost {

    public:

        int getArea() {

            return (width \* height);

        }

};

int main() {

    Rectangle Rect;

    int area;

    Rect.setWidth(5);

    Rect.setHeight(7);

    area = Rect.getArea();

    // Print the area of the object.

    std::cout << "Total area: " << Rect.getArea() << std::endl;

    // Print the total cost of painting

    std::cout << "Total paint cost: $" << Rect.getCost(area) << std::endl;

    return 0;

}

Output:

Total area: 35

Total paint cost: $2450

##### Notes

Despite being supported by C++, **programmers are highly recommended NOT to use multiple inheritance**. Some high level languages, like C# or Java, already removed it. Here’s some reasons:

* Make the code more complex and harder to understand.
* In below example, when C::foo() method is called, which one will be invoked – B::foo() or C::foo()? The compiler doesn’t know, so it triggers error.

#include <iostream>

using namespace std;

class A  {

public:

    void foo() {

        cout << "A::foo()" << endl;

    }

};

class B  {

public:

    void foo() {

        cout << "B::foo()" << endl;

    }

};

class C : public A, public B {

};

int main() {

    C c;

    c.foo();  // Cause ERROR: "request for member ‘foo’ is ambiguous"

    return 0;

}

* We have a better alternative way to achieve similar functionality as multiple inheritance – it’s **composition**. Composition is considered better than inheritance, as it’s more flexible and maintainable (allowing loose coupling and dependencies between classes).

##### Virtual Inheritance

To resolve the above issue, C++ provide a mechanism called virtual inheritance. By using it, a single shared base class instance is created, eliminating the ambiguity.

To inherit virtually, we simply add a keyword virtual before the base class name in the derived class declaration. This indicates that we want to inherit the base class virtually.

Example:

#include <iostream>

using namespace std;

class A {

public:

    int x = 1;

};

class B : virtual public A {

};

class C : virtual public A {

};

class D : public B, public C {

};

int main() {

    D d;

    cout << d.x << endl;

}

Output:

1

#### Hybrid Inheritance



## Polymorphism

The word *polymorphism* means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes that are related by inheritance.

Polymorphism means that **a call to a member function will cause a different function to be executed depending on the type of object** that invokes the function.

Before finding out how polymorphism works, let consider the following WRONG example:﻿

### Example ﻿

#include <iostream>

class Shape {

    protected:

        int width;

        int height;

    public:

        Shape(int w = 0, int h = 0) {

            width = w;

            height = h;

        }

        void area() { // Mistake here!!

            std::cout << "Parent class. Unexpected result!" << std::endl;

        }

};

class Rectangle : public Shape {

    public:

        Rectangle(int w = 0, int h = 0) : Shape(w, h) { }

        void area() {

            int area = width \* height;

            std::cout << "Rectangle area: " << area << std::endl;

        }

};

class Triangle : public Shape {

    public:

        Triangle(int w = 0, int h = 0) : Shape(w, h) { }

        void area() {

            int area = width \* height / 2;

            std::cout << "Triangle area: " << area << std::endl;

        }

};

// Main function for the program

int main() {

    Shape\* rect = new Rectangle(1, 2);

    rect->area();

    Shape\* tri = new Triangle(2, 3);

    tri->area();

    return 0;

}

Output:

Parent class. Unexpected result!

Parent class. Unexpected result!

The reason for the wrong output is that the call of the area() is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called ***early binding*** because the area() is set during the compilation of the program.

Now, let's make a slight modification in our program and precede the declaration ofarea()in the Shape class with the keyword virtual, like this:

virtual void area() {

std::cout << "Parent class. Unexpected result!" << std::endl;

}

With this change, the virtual function area() in the base class WON’T be executed. Instead, the compiler will look at the contents of the pointer shape. Because addresses of objects rec and tri are stored in pointer shape, the respective area() will be called. Finally, we have the following result:

Rectangle area: 2

Triangle area: 3

As seen, each of the child classes has a separate implementation for the function area(). This is how polymorphism is generally used. You have different classes with a function of the same name and even the same parameters, but with different implementations. ﻿This is called *Function Overriding*.

From the example above, you can define polymorphism as "**Polymorphism means writing general code to work with different objects without knowing their exact types**."

### Function Overriding

Function overriding allows **redefinition of base class function in its derived class with same signature**, including number of parameters, datatype of parameters, and return type.

Notes:

* Function overriding is achieved at **run time**.
* It can only be done in **derived class**.
* Overridden functions are in **different scopes**.
* Overridden function can be denoted with keyword override at the end of the function declaration, but it’s not a must.

For example:

#include<iostream>

using namespace std;

class BaseClass

{

public:

    virtual void Display() {

        cout << "This is BaseClass::Display()" << endl;

    }

    void Show() {

        cout << "This is BaseClass::Show()" << endl;

    }

};

class DerivedClass : public BaseClass

{

public:

    // Overridden method

    void Display() override {   // code still works well if "override" is removed

        cout << "This is DerivedClass::Display()" << endl;

    }

};

int main()

{

    DerivedClass dr;

    dr.Show();

    dr.Display();

    BaseClass bs;

    bs.Display();

    BaseClass &bs2 = dr;

    bs2.Display();

}

Output:

This is BaseClass::Show()

This is DerivedClass::Display()

This is BaseClass::Display()

This is DerivedClass::Display()

### Virtual Function

A virtual function is **a function in a base class that is declared using the keyword virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or ***late binding***.﻿

### Pure Virtual Functions

It’s possible to include a virtual function in a base class with no meaningful definition. This requires the **derived class MUST define the overriden function**.

To change the virtual function in the BaseClass to the pure virtual function, add a "= 0" at the end of the function declaration tells the compiler that the function has no body:

virtual void Display() = 0;

### Virtual Destructor

A virtual destructor ensures that derived class destructors are called correctly when deleting objects through base class pointers.

For example:

#include <iostream>

using namespace std;

class Base {

public:

    virtual ~Base() {

        cout << "Base class destructor" << endl;

    }

};

class Derived : public Base {

private:

    int\* ptr;

public:

    Derived() {

        ptr = new int();

    }

    ~Derived() override {

        cout << "Derived class destructor" << endl;

        delete ptr;

    }

};

int main() {

    Base\* derived = new Derived();

    delete derived;

    return 0;

}

Output:

Derived class destructor

Base class destructor

If the base class destructor is not virtual, the derive class destructor will never be called. So no one will free the memory allocated by ptr.

### Others

Typeid

[typeid operator in C++ with Examples - GeeksforGeeks](https://www.geeksforgeeks.org/typeid-operator-in-c-with-examples/)

<http://www.java2s.com/Code/Cpp/Class/Anexamplethatusestypeidonapolymorphicclasshierarchy.htm>

Std::is\_based\_of<>

<https://en.cppreference.com/w/cpp/types/is_base_of>

## Abstraction

Data abstraction refers to, **providing only essential information to the outside world and hiding their background details** (i.e., to represent the needed information in program without presenting the details).

Data abstraction is achieved by using Abstract Data Types (ADTs). ADTs define the data and operations on that data without specifying the implementation. Abstract classes and interfaces are specific types of ADTs.

### Why Helpful?

It’s a technique that **helps separation of interface and implementation**.

Real-life example

Take using a TV as an example. You should know how to turn it on and off, change its channel, adjust its volume, and add external components (speakers, DVD players, etc.). BUT you don’t need to know its internal details, that is, how it receives signals through a cable, how it translates them, and displays them on the screen.

We can say a TV clearly separates its internal implementation from its external interface. So you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

A class can provide JUST ENOUGH public methods to the outside world to play with the functionality of the object and to manipulate object data. That will be much easier for programmers to use it.

In summary, data abstraction helps:

* **Reduce complexity**: Simplify the usage of complex systems 🡪 programmers don’t need to worry about the underlying complexity.
* **Encourage extension**: Promote the use of abstract data types 🡪 easier to maintain and extend the codebase.

### Example

Data abstraction can be obtained by properly using public and private access specifiers.

#include <iostream>

using namespace std;

class Adder {

private:

    // hidden data from outside world

    int total;

public:

    Adder(int i = 0) {

        total = i;

    }

    // interface to outside world

    void addNum(int number) {

        total += number;

    }

    // interface to outside world

    int getTotal() {

         return total;

    };

};

int main() {

   Adder a;

   a.addNum(10);

   a.addNum(20);

   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;

   return 0;

}

### Abstract Classes

**What:**

An abstract class is a class that is **specifically designed to a base class**. It has **AT LEAST ONE pure virtual function**. Then the derived classes (in this case, called *concrete classes*) will provide the implementation for the pure virtual functions.

A special thing about abstract classes that differentiates them from normal base classes is that you **CANNOT declare an object of an abstract class**. However, you can declare pointers and references to an abstract class.

**Why:**

Abstract classes help:

* Allow to create objects of concrete classes, but don’t allow create object of the abstract class itself.
* Provide a way to have both abstract and detailed functionalities 🡪 you can choose to either implement pure virtual functions in concrete classes or reuse virtual functions from the abstract class.

**Example**:

#include <iostream>

// Abstract class

class Shape {

public:

    virtual double area() const = 0;    // Pure virtual function

    void printArea() const {

        std::cout << "Area: " << area() << std::endl;

    }

};

// Concrete subclass

class Rectangle : public Shape {

private:

    double length;

    double width;

public:

    Rectangle(double l, double w) : length(l), width(w) {}

    // Implementation of the pure virtual function

    double area() const override {

        return length \* width;

    }

};

// Concrete subclass

class Circle : public Shape {

private:

    double radius;

public:

    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function

    double area() const override {

        return 3.14159 \* radius \* radius;

    }

};

int main() {

    Rectangle rectangle(5, 4);

    rectangle.printArea();  // Output: Area: 20

    Circle circle(3);

    circle.printArea();     // Output: Area: 28.2743

// Shape shape; // error: cannot declare variable ‘shape’ to be of abstract type ‘Shape’

    return 0;

}

### Interfaces

**What:**

An interface can be represented by an abstract class that **consists ENTIRELY of pure virtual functions**. So the derived classes (in this case, called *concrete classes*) will provide the implementation for ALL pure virtual functions. Otherwise, you’ll get syntax error during compilation.

In addition, an interface **CANNOT has any variable**, but only methods without implementations. And like abstract classes, you **CANNOT declare an object of the interface type**.

**Why:**

* Interfaces provide the highest level of abstraction. This is useful where you want to define a common interface for different implementations without exposing any internal detail.
* Interfaces establish a clear and standardized communication between different parts of a system. They promote loose coupling, making it easier to work on different components independently. This is extremely useful when working on large-scale projects or collaborating with other developers.

## Encapsulation

Encapsulation is **placing the data and the functions that work on that data into a single unit** called a *class*to keep them safe from outside interference and misuse.

Data encapsulation leads to the concept of *data hiding*.

### Why Helpful?

Encapsulation helps:

* **Promote modularity**: Promote modular design and code organization 🡪 improve code readability, maintainability, and reusability.
* **Secure code**: Hide internal implementation details, exposing only the necessary interface to interact with the class 🡪 protect the integrity of the data and reduce the risk of side effects.
* **Code flexibility**: Help to modify the internal implementation details without affecting the code that uses the class.

### Example

Same as Abstraction

# Namespace

## Definition

Namespaces are used to group related entities such as classes, variables and functions into separate named scopes. So entities that would have global scope are grouped into narrower scopes.

Note: Namespaces are not present in C.

## Usages

Namespaces are important for the following reasons:

* Help **prevent naming conflicts** that can occur when multiple libraries or modules are used together. Entities with same names in different namespaces are different.
* Promote encapsulation and modularity. This help **organize code and improve readability**.

## Examples

A namespace definition has following features:

* Namespace declarations appear only at global scope.
* Namespace declarations **can be nested** within another namespace.
* Namespace declarations **don’t have access specifiers**.
* No need to give a semicolon after the closing brace of the definition of namespace.
* We can split the definition of namespace over several units.

**Example 1**: Without class

#include <iostream>

using namespace std;

// first namespace

namespace first\_space {

    void func()

    {

        cout << "Inside first\_space" << endl;

    }

}

// second namespace

namespace second\_space {

    void func()

    {

        cout << "Inside second\_space" << endl;

    }

}

int main() {

    first\_space::func();

    second\_space::func();

    return 0;

}

Output:

Inside first\_space

Inside second\_space

**Example 2**: With class

#include <iostream>

using namespace std;

// Creating a namespace

namespace ns {

    void display();

    class MyClass {

        public:

            void display();

    };

}

void ns::MyClass::display() {

    cout << "ns::MyClass::display()\n";

}

void ns::display() {

    cout << "ns::display()\n";

}

int main() {

    ns::display();

    ns::MyClass myClass;

    myClass.display();

    return 0;

}

Output:

ns::display()

ns::MyClass::display()

## The 'using namespace' Directive

You can **avoid prepending of namespaces** with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace.

Take the above example 1, but this time, we add the using directive before the main() function and call func() from it.

...

using namespace first\_space;

int main() {

    func();

    return 0;

}

Output:

Inside first\_space

## The 'using' Directive

If you don’t want to access the whole namespace, you can **access a particular item within a namespace**.

For example, you can replace std::cout with cout, but cannot replace std::endl with endl in the below code:

#include <iostream>

using std::cout;

int main() {

    cout << "std::endl is used with std!" << std::endl;

    return 0;

}

## Inline Namespace

<https://learn.microsoft.com/en-us/cpp/cpp/namespaces-cpp?view=msvc-170>

## Namespace Aliases

<https://learn.microsoft.com/en-us/cpp/cpp/namespaces-cpp?view=msvc-170>

# Preprocessor

Preprocessors are NOT a part of the compiler, but **a separate step in the compilation process**. In simple terms, a preprocessor is just a **text substitution tool** which instructs the compiler to do required pre-processing before the actual compilation.

All preprocessor commands begin with a hash symbol #. It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.

## Macros

Macros are piece of code which is given some names. Whenever the compiler encounters these names, it replaces them with actual pieces of code. The #define directive is used to define a macro.

#include <iostream>

// macro definition

#define LIMIT 5

int main() {

for (int i = 0; i < LIMIT; i++) {

std::cout << i << "\n";

}

return 0;

}

Output:

0

1

2

3

4

In the above program, when the compiler executes the word LIMIT, it replaces it with 5. The word ‘LIMIT’ in macro definition is called macro template and ‘5’ is macro expansion.

**Notes:**

1. There is no semi-colon (;) at the end of a macro definition.

2. #define vs const

#define A B tells the preprocessor to substitute B wherever it sees A in the code, and it does it **before compiling** the code. It requires **no memory** to store in your program, as it just replaces some text with a literal value, and so, it is as fast as it can get. Moreover, it has **no type and no scope**, so it can be used for any integer value without generating warnings, across files.

A const variable means that once the variable is set, it can't be changed. It doesn't do anything with the preprocessor and it is subject to the normal rules of variables (has scope, take its address, pass it around, use cast it, convert it, etc.).

### Predefined Macros

C defines a number of macros:

|  |  |
| --- | --- |
| **Macro** | **Description** |
| \_\_DATE\_\_ | The current date (at the same of compilation) as a character literal in "MMM DD YYYY" format. |
| \_\_TIME\_\_ | The current time (at the same of compilation) as a character literal in "HH:MM:SS" format. |
| \_\_FILE\_\_ | The current file name as a string literal. |
| \_\_FUNCTION\_\_  (or \_\_func\_\_) | The current function name as a string literal.  For example: getCarName |
| \_\_PRETTY\_FUNCTION\_\_ | The current class name + function name + return type + argument type (if have) as a string literal. Used in C++ only.  For example: void Vehicle::getCarName(int) |
| \_\_LINE\_\_ | The current line number as a decimal constant. |
| \_\_STDC\_\_ | Defined as 1 when the compiler complies with the ANSI standard. |

Let's try the following example:

#include <stdio.h>

int main() {

   printf("File: %s\n", \_\_FILE\_\_ );

   printf("Date: %s\n", \_\_DATE\_\_ );

   printf("Time: %s\n", \_\_TIME\_\_ );

   printf("Function: %s\n", \_\_FUNCTION\_\_ );

   printf("Func: %s\n", \_\_func\_\_ );

   printf("Pretty Function: %s\n", \_\_PRETTY\_FUNCTION\_\_ );

   printf("Line: %d\n", \_\_LINE\_\_ );

   printf("ANSI: %d\n", \_\_STDC\_\_ );

}

Output:

File: main.cpp

Date: Aug 9 2023

Time: 08:55:41

Function: main

Func: main

Pretty Function: int main()

Line: 11

ANSI: 1

### Parameterized Macros

One of the powerful functions of C is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows:

int square(int x) {

return x \* x;

}

We can rewrite above the code using a macro as follows:

#define square(x) ((x) \* (x))

**Note**: The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are NOT allowed between the macro name and open parenthesis. For example:

#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {

printf("Max between 20 and 10 is %d\n", MAX(10, 20));

return 0;

}

Output:

Max between 20 and 10 is 20

## File Inclusion

This type of preprocessor directive tells the compiler to include a file in the source code. There are two types of files which can be included:

#include <stdio.h>

#include "myheader.h"

The first line tells the CPP to get *stdio.h* from System Libraries and add the text to the current source file. The next line tells CPP to get *myheader.h* from the local directory and add the content to the current source file.﻿

## Conditional Compilation

These directives help to skip compilation of some specific part of the program based on some conditions. This can be done with the help of these keywords:

#ifdef DEBUG

/\* Your debugging statements here \*/

#endif

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *DEBUG* flag to the GCC compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

#ifndef MESSAGE

#define MESSAGE "You wish!"

#endif

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

#undef FILE\_SIZE

#define FILE\_SIZE 42

It tells the CPP to undefine existing FILE\_SIZE and define it as 42.

## #pragma

This directive is a special purpose directive, which is used to turn on or off some features. This type of directives is compiler-specific, meaning they vary from compiler to compiler. Some of the #pragma directives are discussed below:

### #pragma startup and #pragma exit

These directives helps specify the functions that are needed to run before program startup (before the control passes to main()) and just before program exit (just before the control returns from main()).

**Note:** The 1st example below will work with MSVC, but not work with GCC compilers or Clang compilers as they don’t supports #pragma startup or exit. However, you can use the code in 2nd example for a similar output on GCC or Clang compilers.

Example 1:

#include <stdio.h>

void func1();

void func2();

#pragma startup func1

#pragma exit func2

void func1()

{

printf("Inside func1()\n");

}

void func2()

{

printf("Inside func2()\n");

}

int main()

{

void func1();

void func2();

printf("Inside main()\n");

return 0;

}

Output:

Inside func1()

Inside main()

Inside func2()

Example 2:

#include <stdio.h>

void func1();

void func2();

void \_\_attribute\_\_((constructor)) func1();

void \_\_attribute\_\_((destructor)) func2();

void func1()

{

printf("Inside func1()\n");

}

void func2()

{

printf("Inside func2()\n");

}

int main()

{

printf("Inside main()\n");

return 0;

}

Output:

Inside func1()

Inside main()

Inside func2()

### #pragma once

It is a non-standard (**not supported by all compilers**) but widely supported preprocessor directive designed to **cause the current source file to be included only ONCE in a single compilation**.

**Example:**

File "grandparent.h"

#pragma once

struct foo {

int member;

};

File "parent.h"

#include "grandparent.h"

File "child.c"

#include "grandparent.h"

#include "parent.h"

In this example, the inclusion of *grandparent.h* in both *parent.h* and *child.c* would ordinarily cause a compilation error. The #pragma once directive helps avoid this by ignoring subsequent inclusions of *grandparent.h*.

**Advantages:**

The most common alternative to #pragma once is to use #define to set an include guard, **the name of which is picked by the programmer to be unique to that file**. For example,

#ifndef GRANDPARENT\_H

#define GRANDPARENT\_H

// contents of grandparent.h

#endif

This is more complicated, possibly less efficient, and prone to error as there are no mechanisms to prevent a programmer accidentally using the same macro name in more than one file.

Using #pragma once instead of include guards will improve compilation speed since it is a higher-level mechanism; the compiler itself can compare filenames without having to invoke the C preprocessor to scan the header for #ifndef and #endif.

### #pragma warn

This directive is used to hide the warning message which are displayed during compilation.

* #pragma warn -rvl: This directive hides those warning which are raised when a function which is supposed to return a value does not returns a value.
* #pragma warn -par: This directive hides those warning which are raised when a function does not uses the parameters passed to it.
* #pragma warn -rch: This directive hides those warning which are raised when a code is unreachable. For example, any code written after the return statement in a function is unreachable.
* ...

## Preprocessor Operators

The C preprocessor offers the following operators to help create macros:

### Macro Continuation *(\)* Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example:

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n")

### Stringize *(#)* Operator

The stringize or number-sign operator (#), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example:

#include <stdio.h>

#define message\_for(a, b) \

printf(#a " and " #b ": We love you!\n")

int main(void) {

message\_for(Carole, Debra);

return 0;

}

Output:

Carole and Debra: We love you!

### Token Pasting *(##)* Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example:

#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {

int token34 = 40;

tokenpaster(34);

return 0;

}

Output:

token34 = 40

It happened so because this example results in the following actual output from the preprocessor:

printf ("token34 = %d", token34);

This example shows the concatenation of token##n into token34 and here we have used both stringize and token-pasting.

# Memory Management – Stack vs Heap

## Memory Layout

When a program is stared, its execuable data is loaded into the memory (RAM). Its memory area will be structured as following:



*Program memory is divided into different segments*

* A text segment for program instructions
* A data segment for global or static variables which are explicitly initialized
* A bss segment for global or static variables which are NOT explicitly initialized
* A stack segment for temporary (or automatic) variables defined in subroutines and functions
* A heap segment for variables allocated during runtime by functions, such as malloc (in C)

### Text Segment (Code Segment)

It’s one of the sections of a program in an object file or in memory, which **contains executable instructions (compiled binary code)**.

Usually, the text segment is **sharable** so that only a single copy needs to be in memory for frequently executed programs, such as text editors, C compiler, shells, and so on. Also, the text segment is often **read-only**, to prevent a program from accidentally modifying its instructions.

### Data Segment (Initialized Data Segment)

It’s a portion of virtual address space of a program, which **contains the global variables and static variables** that are initialized (!= 0) by the programmer.

This segment can be further classified into initialized **read-only** area and initialized **read-write** area.

For example:

* The global string defined by char s[] = "hello world" or static int debug = 1 is stored in initialized **read-write** area.
* The global statement like const char\* string = "hello world" makes the string literal to be stored in **read-only** area and the character pointer variable in **read-write** area.

### BSS (Uninitialized Data Segment)

Uninitialized data segment, often called the "BSS" segment, named after an ancient assembler operator that stood for "block started by symbol."

This segment **contains the global variables and static variables** that are uninitialized or initialized to 0 by the programmer.

For example, a global variable declared int i or static int j would be contained in the BSS segment.

### Stack (Automatic Memory Storage Segment)

Stack is a region of the computer's memory which **stores temporary variables created by each function**. It is a "LIFO" (last in, first out) data structure. Every time a function declares a new variable, it is "pushed" onto the stack. **Then every time a function exits, all of the variables pushed onto the stack by that function, are deleted**. Once a stack variable is deleted, that region of memory becomes available for other stack variables.

Because of being stored temporarily, stack variables are local in nature. This is related to a concept we often saw – variable scope (or local vs global variables).

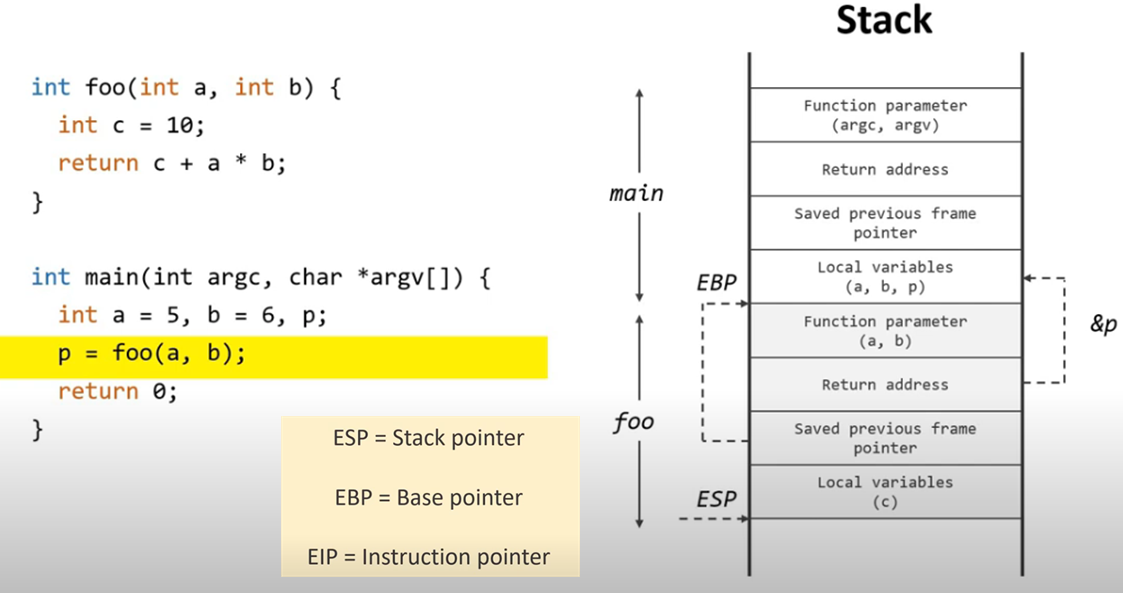
Advantages:

* Data in stack **is automatically managed by OS**. You don't have to allocate memory by hand, or free it once you don't need it any more.
* Because CPU organizes stack memory so efficiently, reading from and writing to stack variables is **very fast**.

Disadvantages:

* There is a **size** **limit** (varies with OS) of variables that can be stored on the stack.

How data is stored in stack:



### Heap (Dynamic Memory Allocation Segment)

Like the stack, the heap is a region of the computer's memory. However, its data is accessible by any function, anywhere in your program. So, heap variables are essentially global in scope.

Advantages:

* It’s a more free-floating region of memory.
* It’s **size is much larger** than stack.

Disadvantages:

* It’s NOT managed automatically for you and NOT as tightly managed by the CPU.

An extremely important note is that once you have allocated memory on the heap, you are responsible for **deallocating that memory once you don't need it any more.** If you forget to do this, your program will have what is known as a [memory leak](#_1ci93xb). That is, memory on the heap will still be set aside (and won't be available to other processes). There is a tool called [valgrind](http://www.valgrind.org/)that can help you detect memory leaks.

* Heap memory is slightly slower to be read from and written to, because one has to use *pointers* to access memory on the heap.

In C/C++, to store variables in heap, you must use built-in functions and operators like malloc(), calloc(), realloc(), or new. And to delete variables after being used, you must use free() or delete.

## When Stack? When Heap?

When stack:

* If you are dealing with relatively **small variables** and only need to **live a short life** as long as the function using them alive.

When heap:

* If you need to allocate a **large block of memory** (e.g., a large array, or a big struct), and you need to keep that variable around a long time (like a **global**).
* If you are not aware in advance how much memory you will need to store particular information in a defined variable, so you want it to **change size dynamically** (e.g., dynamic arrays that can grow or shrink as needed).

## How To Check Size of Memory Layout

In Linux, the size command reports the sizes (in bytes) of the **text segment**, **data segment**, and **BSS** segment.

1. Check the following simple C program:

#include <stdio.h>

int main(void)

{

return 0;

}

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 8 1216 4c0 memory-layout

2. Let us add one global variable in the program, now check the size of bss:

#include <stdio.h>

int global; /\* Uninitialized variable stored in bss\*/

int main(void)

{

return 0;

}

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 12 1220 4c4 memory-layout

3. Let us add one static variable which is also stored in bss:

#include <stdio.h>

int global; /\* Uninitialized variable stored in bss\*/

int main(void)

{

static int i; /\* Uninitialized static variable stored in bss \*/

return 0;

}

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 16 1224 4c8 memory-layout

4. Let us initialize the static variable which will then be stored in the Data Segment:

#include <stdio.h>

int global; /\* Uninitialized variable stored in bss\*/

int main(void)

{

static int i = 100; /\* Initialized static variable stored in DS\*/

return 0;

}

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 252 12 1224 4c8 memory-layout

5. Let us initialize the global variable which will then be stored in the Data Segment:

#include <stdio.h>

int global = 10; /\* initialized global variable stored in DS\*/

int main(void)

{

static int i = 100; /\* Initialized static variable stored in DS\*/

return 0;

}

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 256 8 1224 4c8 memory-layout

# Dynamic Memory Management

Memory in your C++ program is divided into 2 parts: [stack and heap](#_1y810tw). In this section, we’ll focus on the heap and understand how dynamic memory really works in C++.

## In C++

### new

The new operator is used to allocate memory dynamically for any data-type (either a built-in data type or a user-defined data type like a class or struct).

new data-type;

For example, we can define a pointer to type *double* and then request that the memory be allocated at execution time. We can do this using the newoperator with the following statements:

double\* pvalue = new double; // Allocate a memory block

// Pointer pvalue points to the 1st address of this block

**Note**: There are two versions of new:

new unsigned double; // default initialized (only allocate memory)

new unsigned double(); // zero initialized (not only allocate memory,

// but also set initial value to 0.0)

// if the compiler doesn't support this, have to use memset

// More [details](https://cboard.cprogramming.com/cplusplus-programming/79830-does-cplusplus-new-initialise-its-value-zero.html)

**Good practice**:

* The memory may NOT have been allocated successfully, if the heap had been used up. So, it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

double\* pvalue = new double;

if(!pvalue) {

cout << "Error: out of memory." << endl;

exit(1);

}

**new vs malloc():**

|  |  |
| --- | --- |
| **new** | **malloc()** |
| an **operator** which performs an operation. | a **function** that returns and accepts values. |
| calls the **constructors**. So, it doesn't just allocate memory but also constructs objects with initialized values. | cannot call a constructor. |
| **returns the exact data type**. | **returns void\***. So, type-casting is a must-to-do extra step. |
| **throws a *bad\_alloc* exception if memory allocation fails** | just returns a **null pointer**. |
| automatically calculates the size of the object based on its type. | requires manually specifying the size through a parameter. |

In general, it is **recommended to use operator *new* and *delete* in C++ code**, as they provide type safety and automatic object construction.

### delete

At any point, if the variable dynamically allocated is not anymore required, you can free up the memory it occupies in the heap with the delete operator:

delete pvalue; // Release memory pointed to by pvalue

Example 1: Variable

#include <iostream>

using namespace std;

int main() {

double\* pvalue = new double; // Request memory for the variable

\*pvalue = 29494.99; // Store value at allocated address

cout << "Value of pvalue: " << \*pvalue << endl;

delete pvalue; // Free up the memory.

return 0;

}

Output:

Value of pvalue: 29495

Example 2: Array

char\* pvalue = new char[20]; // Request memory for the array

…

delete[] pvalue; // Delete array pointed to by pvalue

**Note**: For multi-dimensional arrays (i.e., char\*\* pvalue), the syntax for releasing memory is the same (e.g., delete[] pvalue, NOT delete[][] pvalue).

Example 3: Object

#include <iostream>

using namespace std;

class Box {

public:

Box() {

cout << "Constructor called!" << endl;

}

~Box() {

cout << "Destructor called!" << endl;

}

};

int main() {

Box\* myBoxArray = new Box;

...

delete myBoxArray;

return 0;

}

Output:

Constructor called!

Destructor called!

**Note**: If you were to allocate an array of 2 Box objects, the constructor would be called 2 times. Similarly, while deleting these objects, destructor will also be called same number of times.

int main() {

Box\* myBoxArray = new Box[2];

delete[] myBoxArray;

return 0;

}

Output:

Constructor called!

Constructor called!

Destructor called!

Destructor called!

**Good practive**:

* Always **set the pointer to null** after deleting the object it points to. And **make sure the pointer is NOT null** before deleting the object. Reasons:
  + It will prevent a *double-delete* error:

// Worst

delete p;

p = NULL;

delete p; // Cause ERROR: "free(): double free detected in tcache 2"

// Best

if (p != NULL) {

delete p;

p = NULL;

}

// Good

delete p;

p = NULL;

delete p; // Don’t cause error

## In C

These are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

|  |  |
| --- | --- |
| **Function** | **Use of Function** |
| void \*malloc(int num) | Allocates requested size of memory and returns a pointer to the first byte of the allocated space. |
| void \*calloc(int num, int size) | Allocates multiple blocks of memory (each of same size), initializes all bytes to zero, and then returns a pointer to the first byte of the allocated space. |
| void free(void \*address) | Deallocate the previously allocated space |
| void \*realloc(void \*address, int newsize) | Change the size of previously allocated space |

### malloc()

The name malloc stands for "memory allocation".

The function malloc() **reserves a block of memory** of specified size and return a pointer of type void which can be casted into pointer of any form. If the space is insufficient, allocation fails and returns NULL pointer.

Syntax:

ptr = (cast-type\*)malloc(byte-size)

Example:

int\* ptr = (int\*)malloc(100\*sizeof(int));

This statement will allocate either 200 or 400 bytes according to size of int (2 or 4 bytes respectively) and the pointer points to the address of the first byte of memory.

### calloc()

The name calloc stands for "contiguous allocation".

The biggest difference between malloc() and calloc() is that malloc() allocates single block of memory whereas calloc() **allocates multiple blocks of memory** (each of same size). Besides, calloc() sets all bytes in the **allocated block to zero.**

Syntax:

ptr = (cast-type\*)calloc(n, element-byte-size);

Example:

float\* ptr = (float\*)calloc(25, sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float.

**Tip**: calloc() is basically malloc() + memset().

int \*ptr = (int\*)malloc(sizeof(int));

memset(ptr, 0, sizeof(int));

### free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

Syntax:

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

**Example #1: Using malloc() and free()**

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

#include <stdio.h>

#include <stdlib.h>

int main() {

int num, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &num);

ptr = (int\*)malloc(num\*sizeof(int));

if(ptr == NULL) {

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements of array: ");

for(i = 0; i < num; ++i) {

scanf("%d", ptr + i);

sum += \*(ptr + i);

}

printf("Sum = %d", sum);

// We can just need free(ptr), but checking NULL is always a good practice

if(ptr != NULL) {

free(ptr);

ptr = NULL;

}

return 0;

}

**Example #2: Using calloc() and free()**

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

#include <stdio.h>

#include <stdlib.h>

int main() {

int num, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &num);

ptr = (int\*)calloc(num, sizeof(int));

if(ptr == NULL) {

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements of array: ");

for(i = 0; i < num; ++i) {

scanf("%d", ptr + i);

sum += \*(ptr + i);

}

printf("Sum = %d", sum);

free(ptr);

return 0;

}

### realloc()

If the previously allocated memory is insufficient, you can change its size using realloc()**without losing the contents** of already allocated memory.

Syntax:

ptr = (cast-type\*)realloc(ptr, newsize);

Here, ptr is reallocated with size of newsize.

Example:

You already have:

int\* ptr = (int\*)malloc(10\*sizeof(int));

Now, if you want to increase the size of memory from 10 to 20 without losing the contents of already allocated memory, use *realloc().*

ptr = (int\*)realloc(ptr, 20\*sizeof(int));

In this case, realloc() will allocate memory for 20 integers somewhere else and then copy the contents of the first 10 locations from here to the new place. It will also de-allocate the existing memory and return a pointer to the new memory.

**Example #3: Using realloc()**

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*ptr, i , n1, n2;

printf("Enter size of array: ");

scanf("%d", &n1);

ptr = (int\*) malloc(n1 \* sizeof(int));

printf("Address of previously allocated memory: ");

for(i = 0; i < n1; ++i)

printf("%u\t",ptr + i);

printf("\nEnter new size of array: ");

scanf("%d", &n2);

ptr = realloc(ptr, n2 \* sizeof(int));

for(i = 0; i < n2; ++i)

printf("%u\t", ptr + i);

return 0;

}

Check out these examples to learn more:

<https://www.programiz.com/c-programming/examples/dynamic-memory-allocation-largest>

<https://www.programiz.com/c-programming/examples/structure-dynamic-memory-allocation>

### memcpy()

It is used to **copy a block of memory from a location to another**.

// Copies "numBytes" bytes from address "from" to address "to"

void\* memcpy(void\* to, const void\* from, size\_t numBytes);

Example:

#include <stdio.h>

#include <string.h>

int main() {

char str1[] = "Geeks";

char str2[] = "Quiz";

puts("str1 before memcpy ");

puts(str1);

// Copies contents of str2 to sr1

memcpy(str1, str2, sizeof(str2));

puts("\nstr1 after memcpy ");

puts(str1);

return 0;

}

Output:

str1 before memcpy

Geeks

str1 after memcpy

Quiz

## Shadow Copy vs Deep Copy

### Shadow Copy

* The copied object simply points to the **same memory locations as the original object**, without creating new memory region. Only reference address to the original object is finally copied.
* If changes are made to the shared memory or pointed-to data in one object, it will affect the other object as well.
* It’s the **default behavior** for copying objects in C++ if a copy constructor or assignment operator is not explicitly defined.

**Example:**

#include <iostream>

using namespace std;

class MyClass {

public:

    int\* data;

    MyClass(int value) {

        cout << "param constructor" << endl;

        data = new int(value);

    }

    // Copy constructor

// Note: Same output result if the below code is removed

    MyClass(const MyClass& other) {

        cout << "copy constructor" << endl;

        data = other.data;      // Shallow copy

    }

    ~MyClass() {

        cout << "destructor" << endl;

        if (data) {

            delete data;

            data = NULL;

        }

    }

};

int main() {

    MyClass obj1(1);

    MyClass obj2 = obj1;

    \*(obj1.data) = 2; // Modifying obj1's data

    std::cout << "obj1.data: " << \*(obj1.data) << std::endl;

    std::cout << "obj2.data: " << \*(obj2.data) << std::endl;

    return 0;

}

Output:

param constructor

copy constructor

obj1.data: 2

obj2.data: 2

destructor

destructor

As a result, modifying obj1.data will also change the value seen through obj2.data.

**Use case:**

* **Performance optimization**: By sharing the same memory or pointers, shallow copy avoids the overhead of duplicating the data.
* **Immutable objects**: Shallow copy is often used when dealing with immutable objects, where the data cannot be modified after creation. Since the data remains constant, sharing the same memory or pointers among multiple instances is safe and efficient.
* **Copy-On-Write (COW)**: Shallow copy is a fundamental technique in COW implementations. It allows multiple instances to initially share the same data, avoiding unnecessary duplication. When one instance tries to modify the shared data, a deep copy is performed to ensure the instance has a separate copy.

### Deep Copy

* The copied object has its own **separate memory**, which is a replica of the memory in the original object. The contents of the original object are duplicated into the new object.
* Changes made to the memory or pointed-to data in one object do not affect the other object, as they now have separate copies of the data. That means deep copy is **recommended if your class has pointer members or any resources requiring special handling during copying**. Otherwise, the default shallow copy behavior provided by C++ may be sufficient.
* Deep copy requires **explicit implementation of copy constructor and copy assignment operator** to ensure proper duplication of the members

**Example:**

#include <iostream>

using namespace std;

class MyClass {

public:

    int\* data;

    MyClass(int value) {

        cout << "param constructor" << endl;

        data = new int(value);

    }

    // Copy constructor

    MyClass(const MyClass& other) {

        cout << "copy constructor" << endl;

        data = new int(\*other.data);      // Deep copy

    }

    ~MyClass() {

        cout << "destructor" << endl;

        if (data) {

            delete data;

            data = NULL;

        }

    }

};

int main() {

    MyClass obj1(1);

    MyClass obj2(obj1);

    \*(obj1.data) = 2; // Modifying obj1’s data

    std::cout << "obj1.data: " << \*(obj1.data) << std::endl;

    std::cout << "obj2.data: " << \*(obj2.data) << std::endl;

    return 0;

}

Output:

param constructor

copy constructor

obj1.data: 2

obj2.data: 1

destructor

destructor

As a result, modifying obj1.data will NOT change the value seen through obj2.data.

**Use case:**

* **Modifiable objects**: By creating separate copies of the data, changes made to one object do not affect others, ensuring data integrity and avoiding unintended side effects.
* **Resource management**: Deep copy is necessary when objects hold resources such as file handles, network connections, or dynamically allocated memory. Each copy should have its own independent resource to properly manage ownership and prevent resource leaks.
* **Polymorphism**: Deep copy can be important when working with polymorphic objects and inheritance hierarchies. It ensures that each object retains its own separate copy of the derived class's data, preventing slicing and preserving the unique behavior of each derived class.
* **Thread safety**: Deep copy is often used in multi-threaded environments to ensure thread safety. Each thread can work with its own copy of the data without contention or data races.

# Common Memory Issues in C/C++

**Great book *Writing Secure Code***:

<https://ptgmedia.pearsoncmg.com/images/9780735617223/samplepages/9780735617223.pdf>

**Great short video series about memory issues:**

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjGAINguks7wyq7TAnHOZGRl>

## Segmentation Faults

A segmentation fault (aka *segfault*) is a common condition that **causes programs to crash**. They are often associated with a file named core. Segfaults are caused by **a program trying to read or write an illegal memory location**.

### Causes

In summary, a segfault occurs when:

* Access **invalid memory** location (NULL, random, out of range, etc.).
* Write a **read-only** location.
* Out of stack space.

For examples:

**1.** Calling memset() as shown below would cause a program to segfault:

memset((char\*)0x0, 1, 100); // void\* memset(void\* ptr, int value, size\_t num);

The above code is using the memset() to set 1 to 100 consecutive bytes, starting from memory address 0x0 (NULL pointer). But **writing to a NULL pointer or any invalid memory location can cause a segfault**.

**2.** The following three cases illustrate the most common types of array-related segfaults:

Case A

// "Array out of bounds" error

int foo[1000];

for (int i = 0; i <= 1000 ; i++) {

foo[i] = i;

}

Array foo is defined for index = 0, 1, 2, ... 999. However, in the last iteration of the for loop, the program tries to access foo[1000]. This will cause a segfault.

Case B

// Illegal memory access because no memory is allocated for foo2

float \*foo, \*foo2;

foo = (float\*)malloc(1000);

foo2[0] = 1.0;

Memory allocation for foo2 has been missed, so foo2 will point to a random location in memory. Accessing foo2[0] will cause a segfault.

**3.** Another common programming error that leads to segfaults is oversight in the use of pointers.

int foo = 0;

scanf("%d", foo); // missing '&' sign

The scanf() expects the address of a variable as its second parameter. The variable foo might be defined at memory location 1000, but the above function call would try to read integer data into memory location 0 according to the definition of foo.

**4.** A segfault will occur when a program attempts to operate on a memory location in a way that is not allowed (e.g., attempts to write a read-only location would result in a segfault).

**5.** Segfaults can also occur when your program runs out of stack space. This may not be a bug in your program, but may be due instead to your shell setting the stack size limit too small.

For example:

void recursiveFunction() {

    int array[10000];       // Allocate a large array on the stack

    recursiveFunction();

}

int main() {

    recursiveFunction();

    return 0;

}

The recursiveFunction() is recursively called without any termination condition. Since each recursive call creates a new stack frame with the array, the stack usage keeps growing until it exceeds the available stack space. As a result, the program used up all the allocated stack space, and it can't allocate any more stack frames. So, a stack overflow occurs, leading to a segmentation fault.

### Preventations

* Always **initialize pointers** (with a valid memory address or NULL) before using them to helps prevent dereferencing uninitialized or invalid pointers.
* After deallocating memory, **set the pointers to NULL**. This practice helps avoid accessing freed or closed memory.
* **Validate all user input** to ensure it fits within expected ranges.
* Utilize **static and dynamic code analysis tools** to analyze the source code and detect potential issues.

### Solutions

#### Finding Out-Of-Bounds Array References

Most Fortran compilers have an option that will insert code to do bounds checking on all array references during runtime. If an access falls outside the index range defined for an array, the program will halt and tell you where this occurs. For most Fortran compilers, the option is -C, or -check followed by a keyword. See your compiler's user guide to get the exact option. Use bounds checking only when debugging, since it will slow down your program.

Some C compilers also have a bound checking option.

#### Checking Shell Limits

As noted in the last example above, some segfaults are not due to bugs in your program, but are caused instead by system memory limits being set too low. Usually it is the limit on stack size that causes this kind of problem. To check memory limits, use the ulimit command in bash or ksh, or the limit command in csh or tcsh. Try setting the stack size higher, and then re-run your program to see if the segfault goes away.

#### Using Debuggers

If you can't find the problem any other way, you might try a debugger. For example, you could use gdb to view the backtrace of a core file dumped by your program. Whenever programs segfault, they usually dump the content of (their section of the memory) at the time of the crash into a core file.

Start your debugger with the command gdb core, and then use the backtrace command to see where the program was when it crashed. This simple trick will allow you to focus on that part of the code.

If using backtrace on the core file doesn't find the problem, you might have to run the program under debugger control, and then step through the code one function, or one source code line, at a time. To do this, you will need to compile your code without optimization, and with the -g flag, so information about source code lines will be embedded in the executable file.

## Buffer Overflow

A buffer overflow (also called **buffer overun**) occurs **when there is more data in a buffer than its capacity**. The extra data ends up overwriting memory other than the memory controlled by the buffer.

### Types and Causes

#### Stack Overflow

A stack overflow occurs when:

* A buffer, which has been declared on the stack, is written to with **more data than it was allocated** to hold.

For example:

void printHello(char\* name) {

   char buf[10];

   strcpy(buf, "hello ");

   strcat(buf, name);

   puts(buf);

}

The stack-allocated buffer can only hold 9 characters (not counting the terminating NUL character). Since the hello string takes up to 6 characters, this means that there is only space for names of 4 characters or less! If the parameter points to a string that has more than 3 characters, this string is still appended to the string already in the buf variable, meaning that the extra characters will be copied into memory that is not assigned to the buffer.

#### Heap Overflow

Heap overflows occur **in heap** memory rather than on the stack. They occur when:

* A buffer – dynamically allocated memory block in the heap – is written to with **more data than it was allocated to hold.**
* Due to memory management vulnerabilities, such as **double-free** or **use-after-free errors**. These vulnerabilities can result in heap corruption and potentially lead to heap overflows.

### Preventatations

* Always allocate buffers with **sufficient size** to hold the data they are intended to store.
* Replace unsafe strinng manipulation functions with **safe variants** (strncat instead of strcat, etc.) that perform bounds checking.
* **Validate all user input** to ensure it fits within the expected bounds.
* Utilize **static and dynamic code analysis tools** to analyze the source code and detect potential issues.

For various methods to prevent buffer overruns: <http://www.informit.com/articles/article.aspx?p=169527>

### Solution

## Memory Leaks

### Causes

### Preventations

### Solutions

Check tutorial "[Personal\Tutorials\C-C++\Analysis Tools.docx](Analysis%20Tools.docx)"

# Data Structure Alignment

## Definition

Data structure alignment is the way data is arranged and accessed in computer memory.

We can break down the concept into 2 parts:

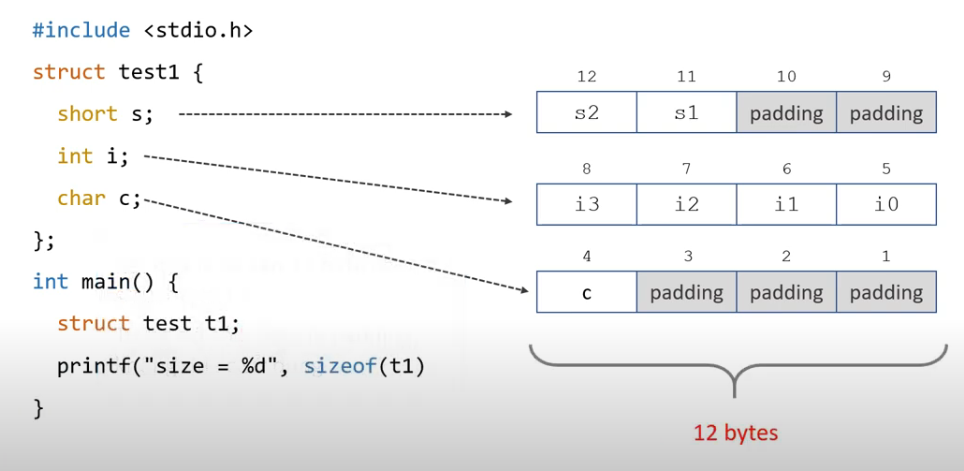
* **Data alignment**: Data alignment means **putting the data in memory** at an address equal to some multiple of the word size. This increases the performance of the system due to the way the CPU handles memory.
* **Data structure padding**: Now, to align the data, it may be necessary to insert some **extra bytes between** the end of the last data structure and the start of the next data structure.

## How It Works

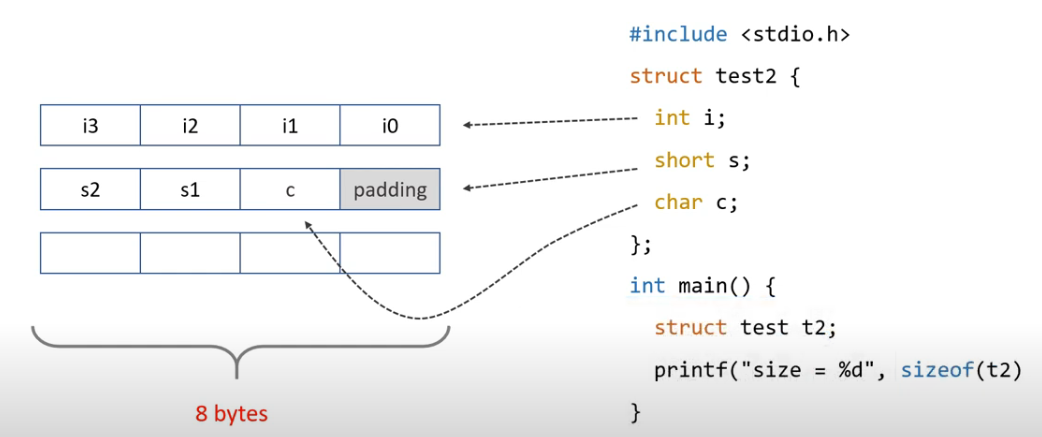
Let’s create two structs. Each containing 1 short, 1 int and 1 char variables, but these variables are in different order.

Tip: Short: 2 bytes, Int: 4 bytes, Char: 1 byte, **Word: 4 bytes**

For the first order: short -> int -> char, the total struct size is 12 bytes.



For the second order: int -> short -> char, the total struct size is 8 bytes.



## Why Padding Is a Must

In each memory cycle, the **processor will access and process data based on word length**. On a 32-bit machine, the word size is 4 bytes.

If the memory is arranged in 4 banks, each of one-byte width, the processor needs to issue 4 memory read cycles to fetch an integer. But we can clearly see that it’ll be much more economical to read all 4 bytes of an integer in one memory cycle. So to take such advantage, the memory will be arranged in one single bank of four-byte width.

In summary, padding bytes helps:

* **Improve memory access performance** by properly aligning data within a structure based on the target architecture.
* **Maintain data integrity** by preventing unintended data corruption caused by misaligned access.

# Cache Locality

*Cache locality*, also known as locality of reference, refers to the **tendency of computer programs to access data elements that are close to each other in memory within a short period of time**.

*You might not know!*

The computer's memory hierarchy consists of different levels of cache (e.g., L1 cache, L2 cache, etc.) and the main memory (RAM). Caches are faster but smaller than the main memory. When the CPU needs to access data, it first checks if the data is available in the cache. If not, it needs to fetch the data from the slower main memory, which takes more time.

Cache locality is beneficial because when data elements are accessed together, they are likely to be stored in nearby locations in the cache. This reduces the number of cache misses (fetching data from the main memory), resulting in faster data access and improved performance.

## Cache Locality and Array

Arrays with good cache locality exhibit 2 types of locality:

* **Temporal Locality**: Refers to reusing the **same data elements repeatedly** within a short period of time. For example, when iterating over an array, adjacent elements are accessed sequentially. Caches are designed to keep recently accessed data in it so that subsequent accesses can be served faster.
* **Spatial Locality**: Refers to accessing **data elements that are physically close to each other in memory**. In arrays, when an element is accessed, the cache fetches a block of data (cache line) containing adjacent elements. If subsequent accesses are made to nearby elements, they are likely to be already present in the cache. This reduces cache misses and improves performance.

On the other hand, poor cache locality, such as accessing elements in a random order or skipping around memory, increases cache misses and leads to slower performance.

Therefore, when designing algorithms or working with arrays, it is beneficial to organize data access patterns to maximize cache locality.

**Example:**

Good cache locality:

#include <iostream>

const int ARRAY\_SIZE = 1000000;

int main() {

    int array1[ARRAY\_SIZE];

    int array2[ARRAY\_SIZE];

    int result[ARRAY\_SIZE];

    // Initialize the arrays

    for (int i = 0; i < ARRAY\_SIZE; i++) {

        array1[i] = i;

        array2[i] = i;

    }

    // Perform the addition

    for (int i = 0; i < ARRAY\_SIZE; i++) {

        result[i] = array1[i] + array2[i];

    }

    // Display a sample result

    std::cout << "Result[100]: " << result[100] << std::endl;

    return 0;

}

Bad cache locality:

#include <iostream>

const int ARRAY\_SIZE = 1000000;

int main() {

    int array1[ARRAY\_SIZE];

    int array2[ARRAY\_SIZE];

    int result[ARRAY\_SIZE];

    // Initialize the arrays

    for (int i = 0; i < ARRAY\_SIZE; i++) {

        array1[i] = i;

        array2[i] = i;

    }

    // Perform the addition with poor cache locality

    for (int i = 0; i < ARRAY\_SIZE; i += 100) {

        result[i] = array1[i] + array2[i];

    }

    // Display a sample result

    std::cout << "Result[100]: " << result[100] << std::endl;

    return 0;

}

In case 1 (good cache locality), we iterate over the entire array sequentially, accessing adjacent elements. This allows the cache to prefetch and store data efficiently.

In case 2 (poor cache locality), we increment the loop index by 100, resulting in non-contiguous memory accesses. This disrupts spatial locality, causing cache misses as the cache needs to fetch data from the main memory for every 100th element.

When we measure the execution time of both cases, we typically observe that case 1 performs significantly faster than case 2. Note that the actual impact of cache locality can vary based on factors like cache size, cache architecture, and the specific hardware platform.

# Exception Handling

Part 1: <https://www.youtube.com/watch?v=mFAaqmj399I>

Part 2: <https://www.youtube.com/watch?v=5369xtKS42s>

One of the advantages of C++ over C is Exception Handling. Exceptions are **run-time abnormal conditions that can arise while a program is running**. There are two types of exceptions: Synchronous and Asynchronous.

Exceptions provide a way to **transfer control from one part of a program to another**. C++ exception handling is built upon three keywords:

* **try** − Identifies a block of code which will be activated for exception handling.
* **throw** − Throws an exception when a problem shows up.
* **catch** − Catches the exception and handles what to do when it occurs.

Note: When an exception is thrown by the throw, the code execution flow will directly jump to the code block defined in the catch, skipping the middle code.

## Throwing Exceptions

Exceptions can be thrown anywhere in a code block using throwstatement (doesn’t need to go along with try and catch). The operand of the throw statement determines a type for the exception.

For example:

double division(int a, int b) {

if(b == 0) {

throw "Division by zero condition!"; // string-type exception

}

return (a/b);

}

## Catching Exceptions

The catchblock following the tryblock catches any exception. You can specify what type of exception you want to catch (corresponding to a specific type of exception thrown by the throw statement). Such type is determined in the parentheses following catch.

try {

// protected code

} catch(ExceptionName &e1) { // [Why &?](https://stackoverflow.com/a/2522356)

// code to handle exception of ExceptionName type

} catch(ExceptionName &e2) {

// code to handle another exception of ExceptionName type

}

You see that:

* There can be **multiple** catch **statements** (each one with a different param type) to catch different types of exceptions in case your tryblock raises more than one exception.
* The syntax for catch is similar to a **regular function with one parameter**. The param type is very important, since the argument type passed by the throw is checked against it, and only if they match, the exception is caught by that handler.

If you want a catch block to handle any type of exception, you can put ... as follows:

try {

// protected code

} catch(...) {

// code to handle any exception

}

The following is an example, which throws a division-by-zero exception:

#include <iostream>

using namespace std;

double division(int a, int b) {

if(b == 0) {

throw "Division by zero!";

}

return (a/b);

}

int main() {

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

} catch (const char\* msg) {

cerr << msg << endl;

}

return 0;

}

Output:

Division by zero!

## Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.

**1) Stop current flow in case of error**

When an exception is thrown by the throw, the code execution flow will directly jump to the code block defined in the catch, **skipping the middle code** (now become useless).

**2) Separation of error handling code from normal code**

In traditional error handling codes, there are always if... else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With exception, the code for error handling becomes **separate from the normal flow**.

**3) Functions/methods can handle any exception they choose**

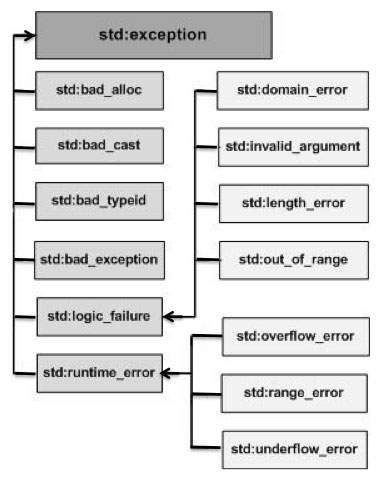
A function can throw many exceptions, but may **choose to handle some** of them. Other exceptions, thrown but not caught, can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

**4) Grouping of error types**

In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, **group exceptions in namespaces or classes**, categorize them according to types.

## Exception Standard Library

C++ provides a list of standard exceptions defined in **<exception>** we can use in our programs. These are arranged in a parent-child class hierarchy as below:

Here is the small description of each of them:

|  |  |
| --- | --- |
| **No** | **Exception & Description** |
| 1 | std::exception  An exception and parent class of all the standard C++ exceptions. |
| 2 | std::bad\_alloc  This can be thrown by new. |
| 3 | std::bad\_cast  This can be thrown by dynamic\_cast. |
| 4 | std::bad\_exception  This is useful device to handle unexpected exceptions in a C++ program. |
| 5 | std::bad\_typeid  This can be thrown by typeid. |
| 6 | std::logic\_error  An exception that theoretically can be detected by reading the code. |
| 7 | std::domain\_error  This is an exception thrown when a mathematically invalid domain is used. |
| 8 | std::invalid\_argument  This is thrown due to invalid arguments. |
| 9 | std::length\_error  This is thrown when a too big std::string is created. |
| 10 | std::out\_of\_range  This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[](). |
| 11 | std::runtime\_error  An exception that theoretically cannot be detected by reading the code. |
| 12 | std::overflow\_error  This is thrown if a mathematical overflow occurs. |
| 13 | std::range\_error  This is occurred when you try to store a value which is out of range. |
| 14 | std::underflow\_error  This is thrown if a mathematical underflow occurs. |

Here is a sample code for handling some std::exception:

try {

    ...

}

catch(const std::invalid\_argument& e) {

    cout << e.what() << endl;

}

catch(const std::out\_of\_range& e) {

    cout << e.what() << endl;

}

catch(const std::exception& e) {

    cout << e.what() << endl;

}

catch(...) {

    cout << "unknown exception" << endl;

}

## Self-Defined Exceptions

You can define your own exceptions by inheriting and overriding std::exceptionclass functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way:

#include <iostream>

#include <exception>

using namespace std;

class MyException : public exception {

public:

const char\* what() {

return "You're driving too fast!\n";

}

};

class MyDriving {

private:

int m\_speed;

public:

int getSpeed() {

return m\_speed;

}

void setSpeed(int speed) {

if(speed >= 150) {

throw MyException();

}

m\_speed = speed;

}

};

int main() {

try {

MyDriving d;

d.setSpeed(160);

cout << "Your valid speed is " << d.getSpeed() << endl;

} catch(MyException &e) {

cout << e.what() << endl;

}

return 0;

}

This would produce the following results:

If d.*setSpeed*(160):

You're driving too fast!

If d.*setSpeed*(140):

Your valid speed is 140

More examples: <https://www.geeksforgeeks.org/exception-handling-c/>

## Nested Exceptions

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

try {

try {

// code here

}

catch (int n) {

throw;

}

}

catch (...) {

cout << "Exception occurred";

}

# Signal Handling

<https://www.tutorialspoint.com/cplusplus/cpp_signal_handling.htm>

# Assertion

## Syntax

In assert.h, assert is a macro function with following syntax:

void assert(int expression);

If the expression equal to 0 (eg, false), a message is written to the standard error device and abort() is called to terminate the program execution.

## Example

Consider the following example:

#include <stdio.h>

#include <assert.h>

// void printNum(int\* inPtr) {

// printf("Inputted num is ");

// if (inPtr == NULL) {

// // terminate the program

// }

// printf("%d\n", \*inPtr);

// }

/\*

Instead of implementing printNum() as above, we can do that using assertion

\*/

void printNum(int\* inPtr) {

printf("Inputted num is ");

// If the expression in assert is false, terminate the program immediately

// and ouput error message to the console

assert(inPtr != NULL);

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

}

int main() {

int a = 1;

int\* b = NULL;

int\* c = NULL;

b = &a;

printNum(b);

printNum(c);

return 0;

}

Output:

Inputted num is 1

Inputted num is

In the above example, by using if, we mean "In my printNum(), I expect people to pass a valid pointer. But if it’s a null pointer, I still know how to handle this situation."

But by using assert, you mean "In my printNum(), I assume that the pointer must be valid. I would be very surprised if it is null and I won’t handle this situation."

## Why Assertions?

**1.** Asserts are **removed from Release mode**. It’s only used to check common errors while writing code, and catch them as soon as possible, in the development phase.

To disable it:

#define NDEBUG

#include <assert.h>

**2.** Asserts will **report failure information** to the client. The specifics of the error message depend on the particular library implementatiom. Something like:

test: main.cpp:9: int main(): Assertion '0==1' failed.

So, if your program crashes in runtime, you will see the exact reason and location of the crash.

**3.** Because asserts are usually macros, you can also get code information about the failing assertion.

**4.** Assert is more semantically clear than if().

## *assert* vs *static\_assert* (C++11)

In C++11, we have static\_assert() which is specially implemented to test code during compile time, while assert() is designed for runtime.

For example:

static\_assert(sizeof(void\*) != 3, "Wrong size"); // can know during compile time

assert(argc == 1); // only know during runtime

assert(ptr != NULL); // only know during runtime

# Recursion

## What Is Recursion?

Recursion is the process in which a **function calls itself** directly or indirectly. Using recursive algorithm, certain problems can be solved quite easily, such as calculating the factorial of a number, generating Fibonacci series, etc.

The idea of using recursion is representing a problem in terms of one or more smaller problems, and add one or more base conditions.

## What Is Base Condition in Recursion?

While using recursion, we need to be careful to **define a base condition (or exit condition)** from the function, otherwise it will go into an infinite loop, leading to **stack overflow error**.

// Factorial of n = 1\*2\*3\*...\*n

#include<iostream>

using namespace std;

unsigned int fact(unsigned int n)

{

    if (n == 0) // base condition

return 1;

    return n\*fact(n-1);

}

// Driver program to test above function

int main()

{

    cout << fact(5);

    return 0;

}

Output:

120

In the above example, base condition for n (n == 0) is defined. When this condition is reached, the recursion function returns a value to stop calling itself.

## Direct vs Indirect Recursion

A function fun is called DIRECT recursive if it calls the same function fun.

A function fun is called INDIRECT recursive if it calls another function, say fun\_new*.*

// An example of DIRECT recursion

void directRecFun()

{

// code....

directRecFun();

// code...

}

// An example of INDIRECT recursion

void indirectRecFun1()

{

// code...

indirectRecFun2();

// code...

}

void indirectRecFun2()

{

// code...

indirectRecFun1();

// code...

}

## Tailed vs Non-Tailed Recursion

A recursive function is tail recursive when recursive call is the last thing executed by the function. Otherwise, it is call non-tailed recursive function.

For example:

// An example of tail recursive function

void print(int n)

{

if (n < 0)

return;

cout << " " << n;

// The last executed statement is recursive call

print(n-1);

}

The tail recursive functions considered better than non-tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple: because the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function’s stack frame is of no use (See [this](https://www.geeksforgeeks.org/tail-call-elimination/) for more details).

**Note**: The fact(int n) function above is non-tail recursive function because the value returned by fact(n-1) is used in fact(n) and call to fact(n-1) is not the last thing done by fact(n)

## How Memory Is Allocated to Different Function Calls in Recursion?

When any function is called from main(), the memory is allocated to it on stack. A recursive function calls itself, the memory for called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Let us take the example how recursion works by taking a simple function.

#include<bits/stdc++.h>

using namespace std;

void printFun(int test)

{

    if (test < 1)

        return;

    else

    {

        cout << test << " ";

        printFun(test-1); // statement 2

        cout << test << " ";

        return;

    }

}

int main()

{

    int test = 3;

    printFun(test);

}

Output :

3 2 1 1 2 3

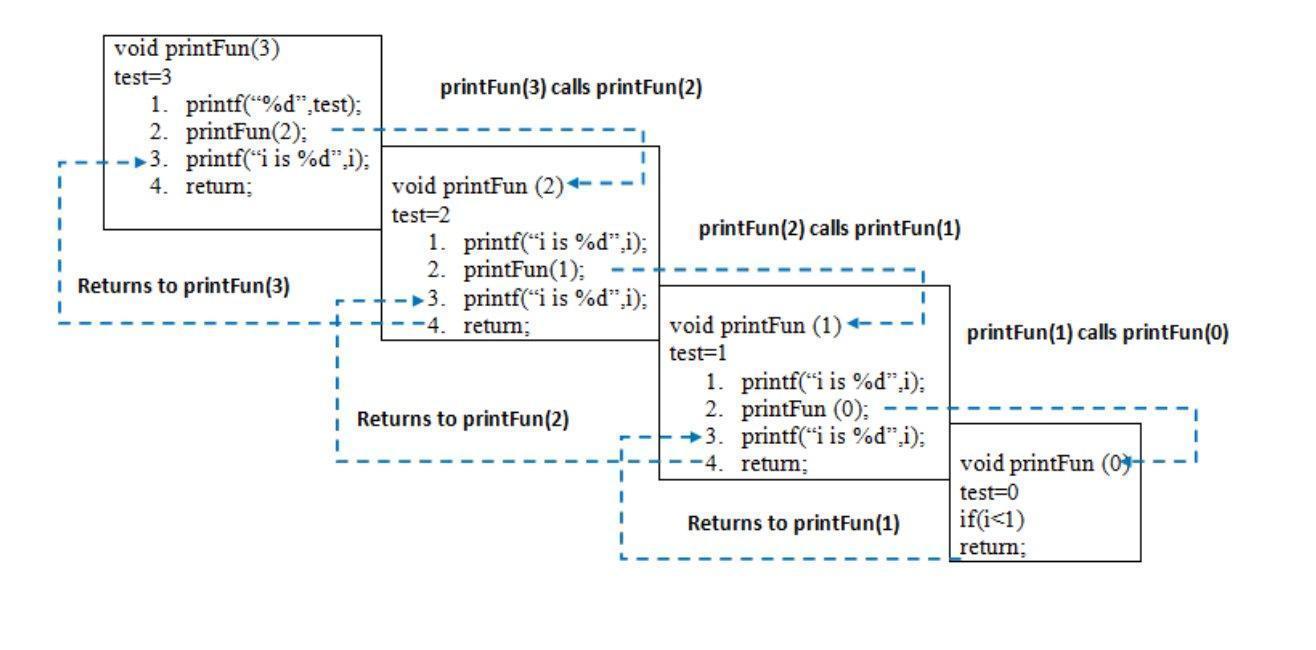
When printFun(3) is called from main(), memory is allocated to printFun(3) and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints ‘3’.

In statement 2, printFun(2) is called and memory is allocated to printFun(2) and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack.

Similarly, printFun(2) calls printFun(1) and printFun(1) calls printFun(0). printFun(0) goes to if statement and it return to printFun(1). Remaining statements of printFun(1) are executed and it returns to printFun(2) and so on.

In the output, value from 3 to 1 are printed and then 1 to 3 are printed.

The memory stack has been shown in below diagram.



## Recursive Programming vs Iterative Programming

Both recursive and iterative programs (using loops) have same problem-solving powers, i.e., **every recursive program can be written iteratively and vice versa**.

**Advantages**

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like calculating the factorial of a number, generating Fibonacci series, etc. For such problems, it is preferred to write recursive code.

**Disadvantages**

Recursive program has greater space requirements than iterative program as all functions will remain in stack until base case is reached. It also has greater time requirements because of function calls and return overhead.

# Function Pointer

Also called *pointers to functions*, but unlike normal pointers (int\*, char\*, etc.), a function pointer points to code, not data. Typically, it **stores the starting address of the block of memory containing all the instructions in a function**. The main benefit of function pointers is that they provide a straightforward mechanism for choosing a function to execute at run-time.

## Syntax

void (\*pf)(int);

1. void is the return type of that function
2. \*pf is the pointer to a function (function pointer)
3. int is the argument type of that function.

**Note**: If we write void \*pf(int), then it will be NO LONGER a function pointer. But it becomes a function called pf with one argument of int type and return value of void\*.

## Example

#include <stdio.h>

void func(int arg) {

    printf("This is func being called and arg is: %d\n", arg);

}

main() {

    void (\*pf)(int);

    pf = &func;     // pf and func must have the same signature

    printf("We're about to call func() using a pointer\n");

    (\*pf)(5);

}

Output:

We're about to call func() using a pointer

This is func being called and arg is: 5

## Notes When Using Function Pointers

**1)** Unlike normal pointers, we do not allocate / de-allocate memory using function pointers.

**2)** A function’s name can also be used to get functions’ address. For example, in the below program, we can remove address operator & in assignment and change function call by removing \*, the program still works.

pf = func; // & removed

pf(5); // \* removed

**3)** Like normal pointers, we can have an array of function pointers.

**4)** Like normal pointers, a function pointer can be passed as an argument and can also be returned from a function. This kind of function pointer is called [callback function](#_147n2zr).

For example, consider the following C program where wrapper() receives a func() as parameter and calls the passed function during runtime.

#include <stdio.h>

void func() {

printf("This is func\n");

}

void wrapper(void (\*func)()) {

    fun();

}

int main() {

    wrapper(func);

    return 0;

}

Output:

This is func

**5)** Many object-oriented features in C++ are implemented using function pointers in C. For example, [virtual functions](#_3j2qqm3). They both solve the same problem: how do we let some code choose the algorithm that is applied. Class methods are another example implemented using function pointers. Refer [this book](http://www.cs.rit.edu/~ats/books/ooc.pdf) for more details.

# Callback Function

## What Is A Callback Function?

A callback is a **reference of a function that is passed as an argument to another function**, which is expected to *call back* (execute) the argument at a given time. So far, we knew that functions can accept only data, but now we know that even a function can be passed.

In C, a callback function is called through a [**function pointer**](#_49x2ik5). In this tutorial, we use the terms *callback* and *function pointer* interchangebly because they’re actually one.

## Examples

**1.** This example shows you the main idea of callback functions:

#include<stdio.h>

void A() {

    printf("I am A\n");

}

void B( void(\*ptr)() ) { // Function B has a function pointer in its param

    ptr();

}

int main() {

    void (\*ptr)() = &A;

    B(ptr); // When B is called, code of A is executed

    // Tip: To make the code neater, we can replace two above statements by:

    // B(A);

    return 0;

}

Output:

I am A

**2.** Another way to use a callback function:

#include <iostream>

typedef void (\*funcPtr)(int);

void myFunc(int value) {

    std::cout << "This is myFunc. Value = " << value << std::endl;

}

int main() {

    funcPtr ptr = myFunc; // Function pointer "ptr" now points to the addess of myFunc

    ptr(1); // So, instead of calling "myFunc" directly, we can do that via "ptr"

    return 0;

}

Output:

This is myFunc. Value = 1

**3**. Another way to implement the example 2:

#include <iostream>

// Set an alias for the function pointer to make the syntax more simple

using cb\_t = void(\*)(int data);

// Declare a callback

cb\_t g\_cb = NULL;

// Set the callback

void registerCb(cb\_t cb) {

    g\_cb = cb;

}

void myFunc(int value) {

    std::cout << "This is myFunc. Value = " << value << std::endl;

}

int main() {

    registerCb(myFunc);

    g\_cb(1);

    return 0;

}

Output:

This is myFunc. Value = 1

**4**. Many built-in functions in standard libraries require a callback as its argument. For example, the [qsort](https://www.programiz.com/cpp-programming/library-function/cstdlib/qsort)() which orders a set of numbers in an descending order:

#include <stdio.h>

#include <stdlib.h> // for qsort()

int compare(const void\* left, const void\* right) {

    return (\*(int\*)right - \*(int\*)left);

}

int main() {

    int (\*cmp) (const void\*, const void\*);

    cmp = &compare;

    int arr[] = {1,2,3,4,5,6,7,8,9};

    int dataSize = sizeof(\*arr);            // Data size of an item in the array

    int arrSize = sizeof(arr)/dataSize;     // Number of items in the array

    // void qsort (void\* base, size\_t num, size\_t size, int (\*com)(const void\*,const void\*));

    qsort(arr, arrSize, dataSize, cmp);

    int i = 0;

    while (i < arrSize) {

        printf("%d ", arr[i]);

        i++;

    }

    return 0;

}

Output:

9 8 7 6 5 4 3 2 1

**5.** Using callback function in class

If the callback function is in a class, it **has to be static**. For example:

#include <iostream>

class SquareCalculator {

public:

    void calc(int num, int (\*cb)(int)) {

        int result = cb(num);

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

    }

    // NOTE: Without "static", error "invalid use of non-static member function"

    static int square(int num) {

        return num \* num;

    }

};

int main() {

    SquareCalculator squareCalculator;

    squareCalculator.calc(5, SquareCalculator::square);

    return 0;

}

Output:

Result: 25

If you don't want to use static function, you can **use functor**:

#include <iostream>

#include <functional>

class SquareCalculator {

public:

    void calc(int num, std::function<int(int)> funcPtr) {

        int result = funcPtr(num);

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

    }

    int square(int num) {

        return num \* num;

    }

};

int main() {

    SquareCalculator squareCalculator;

    squareCalculator.calc(5, [&](int num) {

        return squareCalculator.square(num);

    });

    return 0;

}

# Function Object (Functor)

In C++, a *function object* (or *functor*) is any type that implements operator(). It allows an instance object of a class to be called as if it were a function.

## Example

To create a function object, create a type and implement operator(), such as:

#include <iostream>

class Functor // Note: Functor can be a struct instead of a class

{

public:

    int operator()(int a, int b) {

        return a < b;

    }

};

int main() {

    Functor f;

    int ans = f(1, 2);      // another way: "int ans = f.operator()(1, 2)"

    if (ans) {

        std::cout << "First param is smaller than second param" << std::endl;

    }

    else {

         std::cout << "First param is equal or greater than second param" << std::endl;

    }

}

Output:

First param is smaller than second param

The above example shows how you call a function even though f is an object, not a function. This call looks like a call to a function, but it's actually calling operator() of the Functor type. This similarity is how the term *function object* came about.

## Why Function Object, Not Function Pointer?

Although these concepts can be use interchangeably in many situations, they are totally different – One is a *pointer* and another one is an *object*.

That said, if you have a function that takes a function pointer, you cannot pass in a functor as though it were a function pointer, even if the functor has the same arguments and return value as the function pointer. Likewise, if you have a function that expects a functor, you cannot pass in a function pointer.

Here are significant drawbacks of function pointers:

* *Efficiency* - Function pointers are inefficient when compared with functors. The compiler will often pass them as raw pointers and as such the compiler will struggle to inline the code.
* *State* - Function pointers by themselves are not particularly flexible at storing state between calls to them. Although it is possible, by using a local static variable within the function, there is only ever one global state for the function itself and as such this static variable must be shared. Furthermore, this static variable will not be thread-safe, unless the appropriate thread synchronisation code is added. Thus it can lead to bottlenecks or even race conditions in multithreaded programs.

#include <iostream>

#include <vector>

Struct {

    int sum;

    void operator()(int element) { sum += element; }

} functor;

int main() {

    std::vector<int> vec1 = {1, 2, 3};

    std::vector<int> vec2 = {4, 5, 6};

    functor.sum=0;

    functor = std::for\_each(vec1.begin(), vec1.end(), functor);

    functor = std::for\_each(vec2.begin(), vec2.end(), functor);

    std::cout<<"The sum of all the elements is: "<<functor.sum<<std::endl;

}

Output:

The sum of all the elements is: 21

* *Templates* - Function pointers do not play with templates if there are multiple signatures of the function in your code. A solution is to use function pointer casting, which leads to difficult and ungainly syntax.

The better solution is to using a function object. For example:

// Using function pointer

std::vector<int> find\_matching\_numbers (std::vector<int> vec, bool (\*pred)(int)) {

    std::vector<int> ret\_vec;

    std::vector<int>::iterator start = vec.begin();

std::vector<int>::iterator end = vec.end();

    while(start != end) {

        if(pred(\*start)) {

            ret\_vec.push\_back(\*start);

        }

        ++start;

    }

}

// Using function object

template <typename T>

std::vector<int> find\_matching\_numbers(std::vector<int> vec, T pred) {

    std::vector<int> ret\_vec;

    std::vector<int>::iterator start = vec.begin();

std::vector<int>::iterator end = vec.end();

    while(start != end) {

        if(pred(\*start)) {

            ret\_vec.push\_back(\*start);

        }

        ++start;

    }

}

* *Adaptation* - Function pointers have fixed parameter types and quantities. Thus they are not particularly flexible when external functions with differing parameter types could be used. Although adapting the function pointers (by wrapping the external functions with hard-coded parameters) is possible, it leads to poor flexibility and bloated code.

In other words, function pointers are limited because functions must be fully specified at compile time. For example, you're writing a mail program to view an inbox, and you'd like to give the user the ability to sort the inbox on different fields (to, from, date, etc). You might try using a sort routine that takes a function pointer capable of comparing the messages, but there's one problem – there are a lot of different ways you might want to compare messages. You could create different functions that differ only by the field of the message on which the comparison occurs, but that limits you to sorting on the fields that have been hard-coded into the program. It's also going to lead to a lot of if-then-else blocks that differ only by the function passed into the sort routine.

The solution to these problems is to using a function object. For example:

class Message {

    public:

        std::string getHeader (const std::string& header\_name) const;

};

class MessageSorter {

    private:

        std::string \_field;

    public:

        MessageSorter (const std::string& field) : \_field(field) {}

// Functor

        bool operator() (const Message& lhs, const Message& rhs) {

            return lhs.getHeader(\_field) < rhs.getHeader(\_field);

        }

};

int main() {

    // ...

    std::vector<Message> messages;

    MessageSorter comparator;

    sort(messages.begin(), messages.end(), comparator);

    return 0;

}

## STL Function Object

An obvious advantage of function objects over function pointers that we didn’t directly mentioned in the above section is here. The Standard Template Library (STL) often uses function objects and provides several function objects (in the <[functional](http://www.cplusplus.com/reference/functional/)> header file) that are very helpful. It provides three types of template function objects:

* Generator
* Unary function
* Binary function

### std::function (C++11)

It is a STL *class* that provides a very convenient wrapper to a conventional function, a functor or a lambda expression.

**Syntax**

std:function<return-type(list-of-parameter-types)> instance = a function / functor / lambda

**Example**

If you want to store several functions, functors or lambda expressions in a vector, you could write something like this:

#include <functional>

#include <iostream>

#include <string>

#include <vector>

void execute(const std::vector<std::function<void()>>& fs) {

   for (auto& f : fs) {

      f();

   }

}

void func() {

   std::cout << "I'm a function" << std::endl;

}

class Functor {

public:

   void operator()() const {

     std::cout << "I'm a functor" << std::endl;

   }

};

int main() {

   // Type of vector is a std::function

   std::vector<std::function<void()>> f;

   // Function

   f.push\_back(func);

   // Functor

   Functor functor;

   f.push\_back(functor);

   // Lambda

   f.push\_back([] () {

      std::cout << "I'm a lambda expression" << std::endl;

   });

   // Member function

   ...

   execute(f);

   return 0;

}

Output:

I'm a function

I'm a functor

I'm a lambda expression

### std::bind (C++11)

It is a STL *function* that returns a std::function object that binds a set of arguments to a function.

**Syntax**

std::bind(function-name, list-of-parameter-value);

**Example**

Consider the above example, we store plainFunc() which takes no parameter to std::function. What if this function takes one or several parameters? Well, we have to use std::bind().

void plainFuncWithParam(const char\* str) {

   std::cout << "I'm a plain function with param: " << str << std::endl;

}

std::function<void()> plainFuncWithParam\_obj = std::bind(plainFuncWithParam, "str");

f.push\_back(plainFuncWithParam\_obj);

execute(f);

Output:

I'm a plain function with param: str

As you can see, the std::bind() receives a pointer to a function (it also can be a lambda expression or a functor) and receives a list of parameters that pass it to the function. As result, std::bind() returns a new function object with a different prototype because all the parameters of the function were already specified.

### std::placeholders (C++11)

It is a STL *namespace* that helps to manipulate the position and number of arguments to be used by the function. They are represented by \_1, \_2, \_3…

Comparing to using default parameters, std::placeholders is much more flexible.

**Example**

#include <iostream>

#include <functional>

void substract(int a, int b) {

    std::cout << a - b << std::endl;

}

int main() {

    // \_1 is for first parameter when calling f1

    // 2 is assigned to b

    auto f1 = std::bind(substract, std::placeholders::\_1, 2);

    // 2 is assigned to a

    // \_1 is for first parameter when calling f2

    auto f2 = std::bind(substract, 2, std::placeholders::\_1);

    // Call of modified functions

    f1(10);

    f2(10);

    return 0;

}

Output:

8

-8

# Lambda Expression (C++11)

*Lambda expression* (or *lambda function* or *lambda*) is a convenient way of defining anonymous function and [function object](#_23ckvvd) (a *closure*) right at the location where it is called or passed as an argument to a function. Therefore, lambdas are commonly used for short snippets of code that are not going to be reuse and not worth naming.

## Syntax

In its simplest form, a lambda expression can be defined as follows:

[ capture-clause ] (parameter-list) -> return-type

{

definition of expression

};

**Parameter list**

Like of conventional functions, parameter list is **optional**. You can omit it if you want a function taking zero arguments.

**Return type**

Generally return type in lambda expression are evaluated by compiler itself and we don’t need to specify that explicitly. This means the -> return-type part **can be ignored**. However, when you have more than one return statement, compiler can’t make out the return type and we need to specify that. In addition, if you want your code easier to understand, you can specify the return type.

**Capture clause**

You will always be able to identify a lambda expression with the presence of a []. It also gives lambda expressions a special capability – having **access to variables from the enclosing scope**. We have three ways to do that:

[&] : capture all external variable by reference

[=] : capture all external variable by value

[a, &b] : capture a by value and b by reference

A lambda with empty capture clause [] can access only those variable which are local to it.

## Examples

**1.** A lambda expression with no captured variable, no parameter and no explicit return type:

#include <iostream>

using namespace std;

int main() {

auto lambda = []() {

cout << "Code within a lambda expression" << endl;

};

lambda();

}

**2.** A lambda expression with no captured variable, two parameters and a return type. The return type is deduced, but if we want to make thing clear, we can add -> int at the end of the declaration:

#include <iostream>

using namespace std;

int main() {

auto sum = [](int x, int y) {

return x + y;

};

cout << sum(5, 2) << endl;

}

**3.** The following code use std::count\_if, included in the Standard Templates Library (STL), to count the number of elements in an int vector that are greater than 5. The call to std::count\_if specifies isGreaterThan5 as the user-defined function object that defines the condition to be satisfied if an element is to be counted. It is possible to simplify the code by **using a lambda expression to specify the user-defined function object**.

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

bool isGreaterThan5(int value) {

return (value > 5);

}

int main() {

vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };

auto greaterThan5\_count = count\_if(numbers.begin(), numbers.end(), isGreaterThan5);

cout << "The number of elements greater than 5 is: "

<< greaterThan5\_count << "." << endl;

}

is similar to:

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main() {

vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };

auto greatThan5\_count = count\_if(numbers.begin(), numbers.end(), [](int x) {

return (x > 5);

});

cout << "The number of elements greater than 5 is: "

<< greaterThan5\_count << "." << endl;

}

**4.** Another excellent example of how lambda expressions can **simplify code** is their use with the std::for\_each function:

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

void printNumber(int y) {

cout << y << endl;

}

int main() {

vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };

for\_each(numbers.begin(), numbers.end(), printNumber);

}

is similar to:

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main() {

vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };

for\_each(numbers.begin(), numbers.end(), [] (int y) {

cout << y << endl;

});

}

**5.** It’s time to **capture outscope variables**. In the following example, we capture variable divisor **by value** by putting it in the []. If we want to capture all variables by value in the main(), we can use [=].

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main() {

// The user would introduce different values for divisor

int divisor = 3;

vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };

for\_each(numbers.begin(), numbers.end(), [divisor] (int y) {

if (y % divisor == 0)

{

cout << y << endl;

}

});

}

**6.** A different version of the above example. But this time, we want to take sum of numbers divisible by the divisor. And to make this sum valid outside of the lambda expression, we **capture it by reference**:

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

int main() {

int sum = 0;

int divisor = 3;

vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };

for\_each(numbers.begin(), numbers.end(), [divisor, &sum] (int y) {

if (y % divisor == 0)

{

cout << y << endl;

sum += y;

}

});

cout << sum << endl; // thanks to reference, sum has valid value

}

## Generic Lambda Expressions (C++14)

C++14 allows the use of auto in the **parameter list** of lambda expressions. For example:

auto identity = [](auto x) {

return x;

};

int num = identity(3);

std::string str = identity("foo");

# Type Casting

**Converting an expression of a given type into another type** is known as type-casting. We have some ways to type cast:

## Implicit Conversion

Implicit conversions **do NOT require any operator**. They are automatically performed when a value is copied to a compatible type.

Example 1:

short a = 2000;

int b;

b = a;

Here, the value of *a* has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a **standard conversion**. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...) and some pointer conversions.

Converting to int from some smaller integer type, or to double from float is known as *promotion*, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

* If a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., -1 becomes the largest value representable by the type, -2 the second largest, ...).
* The conversions from/to bool consider false equivalent to *zero* (for numeric types) and to nullptr(for pointer types); true is equivalent to all other values and is converted to the equivalent of 1.
* If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes *undefined behavior*.
* Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is *implementation-specific* (and may not be portable).

Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

* *Null pointers* can be converted to pointers of any type
* Pointers to any type can be converted to void pointers.
* Pointer *upcast*: pointers to a derived class can be converted to a pointer of an *accessible* and *unambiguous* base class, without modifying its const or volatile qualification.

Example 2:

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

class A {};

class B { public: B (A a) {} };

A a;

B b = a;

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore, implicit conversions from A to B are allowed.

## Explicit Conversion

C++ is a strong-typed language. Many conversions, especially those that imply a different interpretation of the value, require an explicit conversion. We have two notations for explicit type conversion:

* **Functional casting**
* **C-like casting**

short a = 2000;

int b;

b = (int)a; // C-like cast notation

b = int(a); // functional notation

The functionality of these explicit conversion operators is **enough for most needs with fundamental data types**.

However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:

// class type-casting

#include <iostream>

using namespace std;

class CDummy

{

float i, j;

};

class CAddition

{

int x, y;

public:

CAddition(int a, int b) {

x = a;

y = b;

}

int result() {

return x + y;

}

};

int main()

{

CDummy d;

CAddition \*pAdd;

pAdd = (CAddition\*)&d; // Cause runtime error

cout << pAdd->result();

return 0;

}

The program declares a pointer to CAddition, but then it is assigned to a reference to an object of another incompatible type using explicit type-casting: pAdd = (CAddition\*)&d

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member result() will produce either a run-time error or a unexpected result.

## Type Casting Operators

In order to control these types of conversions between classes, C++ provides four new specific casting operators: dynamic\_cast, reinterpret\_cast, static\_cast and const\_cast. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

dynamic\_cast <new\_type> (expression)

reinterpret\_cast <new\_type> (expression)

static\_cast <new\_type> (expression)

const\_cast <new\_type> (expression)

### dynamic\_cast

dynamic\_cast can be **used only with pointers and references to objects**. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

**Usecase 1:**

Therefore, dynamic\_cast is **always successful when we cast a class to one of its base classes**:

class CBase { };

class CDerived: public CBase { };

CBase b;

CDerived d;

CBase\* pB = dynamic\_cast<CBase\*>(&d); // right: derived-to-base

CDerived\* pD = dynamic\_cast<CDerived\*>(&b); // wrong: base-to-derived. Only right when polymorphic

**Usecase 2:**

The second conversion in above code would produce a compilation error since base-to-derived conversions are not allowed with dynamic\_cast unless the base class is **polymorphic**.

When a class is polymorphic, dynamic\_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:

// dynamic\_cast

#include <iostream>

#include <exception>

using namespace std;

class CBase {

virtual void dummy() {}

};

class CDerived: public CBase {

int a;

};

int main() {

try {

CBase \*pBa = new CDerived;

CBase \*pBb = new CBase;

CDerived \*pD;

pD = dynamic\_cast<CDerived\*>(pBa); // right

if (pD == 0) {

cout << "Null pointer on first type-cast" << endl;

}

pD = dynamic\_cast<CDerived\*>(pBb); // wrong

if (pD == 0) {

cout << "Null pointer on second type-cast" << endl;

}

} catch (exception& e) {

cout << "Exception: " << e.what();

}

return 0;

}

Output:

Null pointer on second type-cast

The code tries to perform two dynamic casts from pointer objects of type *CBase\** (pBa and pBb) to a pointer object of type CDerived\*, but only the first one is successful. Notice their respective initializations:

CBase \*pBa = new CDerived;

CBase \*pBb = new CBase;

Even though both are pointers of type CBase\*, pBa points to an object of type CDerived, while pBb points to an object of type CBase. Thus, when their respective type-castings are performed using dynamic\_cast, pBa is pointing to a full object of class CDerived, whereas pBb is pointing to an object of class CBase, which is an incomplete object of class CDerived.

**When dynamic\_cast cannot cast a pointer because it is not a complete object of the required class, it returns a null pointer to indicate the failure**. If it’s used to convert to a reference type and the conversion is not possible, an exception of type bad\_cast is thrown instead.

dynamic\_cast can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (void\*).

**Compatibility note**:

dynamic\_cast requires the Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This must be enabled for runtime type checking using dynamic\_cast to work properly.

### static\_cast

**Usecase 1:**

static\_cast can **perform conversions between pointers to related classes**, **not only from the derived class to its base, but also from a base class to its derived**. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of dynamic\_cast is avoided.

class CBase {};

class CDerived: public CBase {};

CBase \*pB = new CBase;

CDerived \*pD = static\_cast<CDerived\*>(pB); // base-to-derived

This would be valid, although pB would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

**Usecase 2:**

static\_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

double d = 3.14159265;

int i = static\_cast<int>(d);

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

### reinterpret\_cast

reinterpret\_cast **converts any pointer type to any other pointer type, even of unrelated classes**. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can **also cast pointers to or from integer types**. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by reinterpret\_cast but not by static\_cast are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

class A {};

class B {};

A \*a = new A;

B \*b = '<B\*>(a);

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

### const\_cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

// const\_cast

#include <iostream>

using namespace std;

void print(char\* str)

{

cout << str << endl;

}

int main() {

const char\* c = "sample text";

print ( const\_cast<char\*> (c) );

return 0;

}

Output:

sample text

## typeid

It allows us to check the type of an expression:

typeid(expression)

This operator returns a reference to a constant object of type type\_info that is defined in the standard header file *<*typeinfo*>.* This returned value can be compared with another one using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

// typeid

#include <iostream>

#include <typeinfo>

using namespace std;

int main() {

int \*a, b;

a = 0; b = 0;

if (typeid(a) != typeid(b))

{

cout << "a and b are of different types:\n";

cout << "a is: " << typeid(a).name() << '\n';

cout << "b is: " << typeid(b).name() << '\n';

}

return 0;

}

Output:

a and b are of different types:

a is: int \*

b is: int

When typeid is applied to classes, it uses the RTTI to keep track of the type of dynamic objects. When *typeid* is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

// typeid, polymorphic class

#include <iostream>

#include <typeinfo>

#include <exception>

using namespace std;

class CBase {

virtual void f(){}

};

class CDerived : public CBase {};

int main() {

try {

CBase\* a = new CBase;

CBase\* b = new CDerived;

cout << "a is: " << typeid(a).name() << '\n';

cout << "b is: " << typeid(b).name() << '\n';

cout << "\*a is: " << typeid(\*a).name() << '\n';

cout << "\*b is: " << typeid(\*b).name() << '\n';

} catch (exception& e) {

cout << "Exception: " << e.what() << endl;

}

return 0;

}

Output:

a is: class CBase \*

b is: class CBase \*

\*a is: class CBase

\*b is: class CDerived

**Note**: The string returned by member name of type\_info depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.

Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class CBase\*). However, when typeid is applied to objects (like \*a and \*b) typeid yields their dynamic type (i.e. the type of their most derived complete object).

If the type typeid evaluates is a pointer preceded by the dereference operator (\*), and this pointer has a null value, typeid throws a bad\_typeid exception.

The compiler in the examples above generates names with type\_info::name that are easily readable by users, but this is not a requirement: a compiler may just return any string.

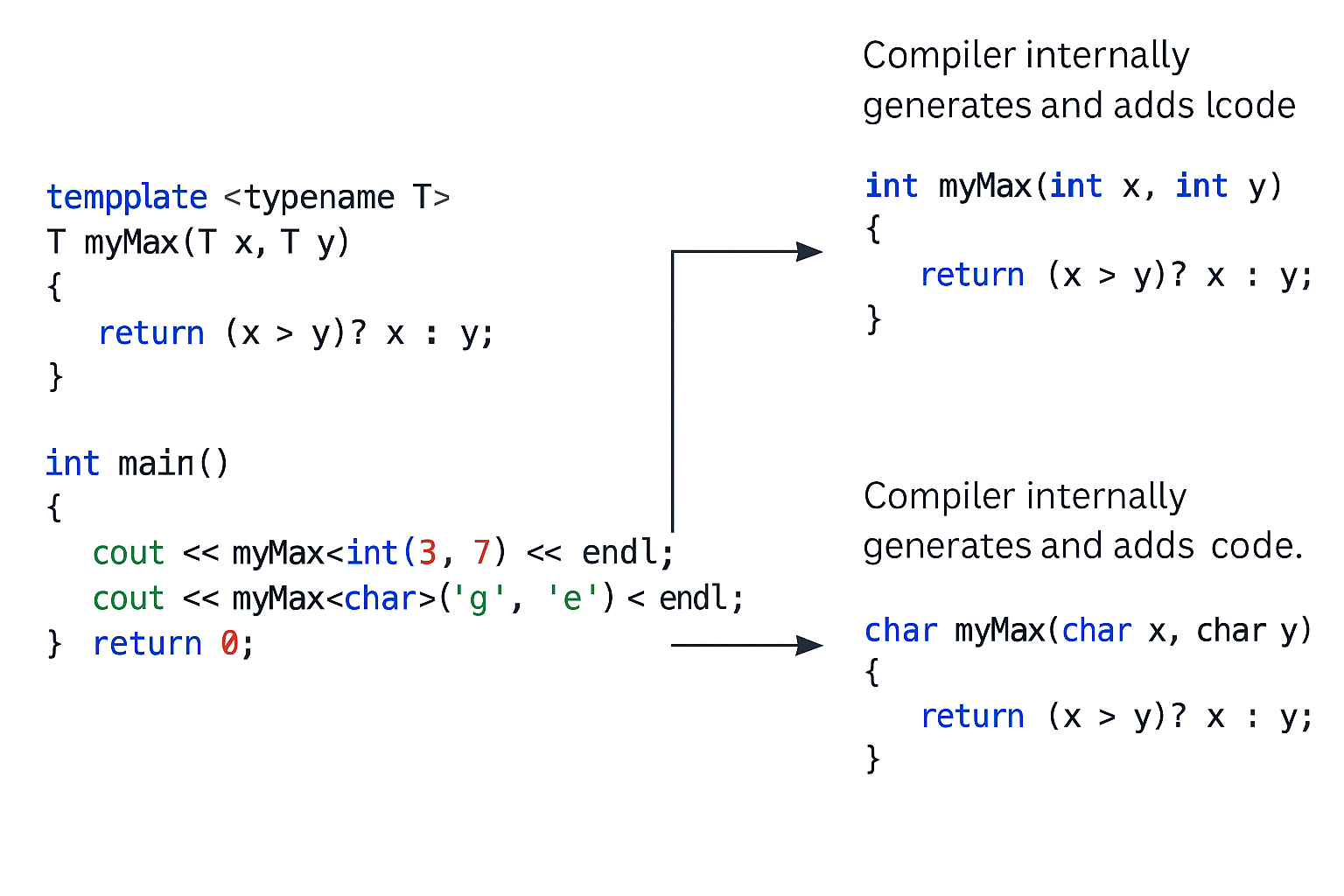
# Template

A template is a simple and powerful tool in C++. The simple idea is to **pass data type as a parameter to a function or class so that we don’t need to write the same code for different data types**.

C++ adds two new keywords to support templates: templateand typename (this can be replaced by class).

## How Templates Work?

Templates are **expanded at compile time**. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple; source code contains only one function/class, but compiled code may contain multiple copies of such function/class.



## Function Templates

Using a template, we can write a generic function for different data types. Typical examples are sort(), max(), min(), printArray().

**Example 1 – Finding max number of any type:**

#include <iostream>

using namespace std;

// One function works for all data types,

// even for user-defined types if operator '>' is overloaded

template <typename T>

T myMax(T x, T y) {

   return (x > y)? x: y;

}

int main() {

  cout << myMax<int>(3, 7) << endl;         // int

  cout << myMax<double>(3.0, 7.0) << endl;  // double

  cout << myMax<char>('g', 'e') << endl;    // char

  return 0;

}

Output:

7

7

g

**Example 2 – Implementing**[**Bubble Sort**](http://www.geeksforgeeks.org/bubble-sort/)**using templates:**

#include <iostream>

using namespace std;

template <typename T>

void bubbleSort(T arr[], int n) {

    for (int i = 0; i < n - 1; i++) {

        for (int j = n - 1; i < j; j--) {

            if (arr[j] < arr[j - 1]) {

               swap(arr[j], arr[j - 1]);

            }

         }

    }

}

int main() {

    int arr[5] = {10, 50, 30, 40, 20};

    bubbleSort(arr, 5);

    cout << "Sorted array: ";

    for (int i = 0; i < 5; i++) {

        cout << a[i] << " ";

    }

    cout << endl;

  return 0;

}

**Output**:

Sorted array: 10 20 30 40 50

## Class Templates

Like function templates, class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

**Example 1 – Implementing Array class of any type:**

#include <iostream>

using namespace std;

// class declaration

template <typename T>

class Array {

public:

    Array(T arr[], int s);

    void print();

private:

    T \*ptr;

    int size;

};

// class definition

template <typename T>

Array<T>::Array(T arr[], int s) {

    ptr = new T[s];

    size = s;

    for(int i = 0; i < size; i++) {

        ptr[i] = arr[i];

    }

}

template <typename T>

void Array<T>::print() {

    for (int i = 0; i < size; i++) {

        cout << " " << \*(ptr + i);

    }

    cout << endl;

}

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    Array<int> a(arr, 5);

    a.print();

    return 0;

}

Output:

1 2 3 4 5

**Example 2 – Specifying default value for template arguments:**

Like normal parameters, we can specify default arguments to templates. For example:

#include<iostream>

using namespace std;

template<typename T, typename U = char>

class A {

public:

    T x;

    U y;

    A() {   cout << "Constructor called" << endl;   }

};

int main() {

   A<char> a;  // This will call A<char, char>

   return 0;

}

## Template Specialization

Sometimes, a generic template implementation isn't suitable for all types. Specialization lets you **provide a custom implementation for specific types**.

### Full Specialization

You provide a **completely** different implementation for a specific type or set of types.

**Example 1 – Function to print string value for different data type**

Here, we provide a completely different implementation for bool and std::string. So, bool will have its own logics that are not same as std::string.

Notice the syntax template <>. This signifies that we are not introducing new template parameters; we are providing a specialization for a fully-specified set of arguments.

#include <string>

#include <iostream>

// Generic template

template <typename T>

std::string stringify(T value) {

std::cout << "This is Generic template" << std::endl;

// Just convert to string with to\_string()

return std::to\_string(value);

}

// Specialization template for bool

template <>

std::string stringify<bool>(bool value) {

std::cout << "This is Specialization template for bool" << std::endl;

// TRUE is "true", FALSE is "false"

return value ? "true" : "false";

}

// Specialization template for string

template <>

std::string stringify<std::string>(std::string value) {

std::cout << "This is Specialization template for string" << std::endl;

// Do nothing because value is already a string

return value;

}

int main() {

std::cout << stringify(123) << std::endl; // Uses generic template

std::cout << stringify(true) << std::endl; // Uses specialized template

std::cout << stringify(std::string("abc")) << std::endl; // Uses specialized template

// std::cout << stringify"abc") << std::endl; // Will cause compilation ERROR because "abc" is const char\*, not std::string

}

### Partial Specialization

Note: This is not available for function templates.

You can specialize a class template for a **category of types** (not any specific type), like all pointer types or all std::vector<T> types.

**Exammple 1 – A class that behaves differently for pointer types.**

Here, we provide an implementation for any pointer type T\*. It's "partial" because we've specified that it must be a pointer, but the type it points to (T) is still a generic template parameter. All pointer types (int\*, double\*, char\*, etc.) will be handled with the same logics.

#include <iostream>

// Primary template

template <typename T>

class TypeInfo {

public:

static const char\* name() { return "some type"; }

};

// Partial specialization for all pointer types T\*

template <typename T>

class TypeInfo<T\*> {

public:

static const char\* name() { return "pointer type"; }

};

int main() {

std::cout << TypeInfo<int>::name() << std::endl;

std::cout << TypeInfo<double\*>::name() << std::endl;

}

## Kinds of Template Parameters

Templates can accept more than just types.

### Non-Type Template Parameters (NTTPs)

You can pass compile-time constant values as template arguments. This is useful for things like fixed-size arrays or compile-time configuration.

**Example 1 – Creating a fixed-size buffer class**:

#include <array>

#include <iostream>

template <typename T, std::size\_t N>

class FixedSizeBuffer {

private:

std::array<T, N> data;

public:

std::size\_t size() const { return N; }

// ... other methods

};

int main() {

FixedSizeBuffer<int, 1024> network\_buffer;

std::cout << "Buffer size: " << network\_buffer.size() << std::endl;

}

### Template Template Parameters

You can **pass a template itself as an argument to another template**. This allows you to abstract over container types.

**Example 1 – Creating a generic class that can use any standard-like container:**

#include <vector>

#include <list>

#include <iostream>

// T is a type, Container is a template that takes one type argument

template <typename T, template <typename> class Container>

class GenericCollection {

private:

Container<T> coll;

public:

void add(const T& value) { coll.push\_back(value); }

// ...

};

int main() {

// Using std::vector

GenericCollection<int, std::vector> vec\_collection;

vec\_collection.add(1);

// Using std::list (Note: std::list has more template parameters, but they have defaults)

// GenericCollection<int, std::list> list\_collection; // This might require a using alias for more complex containers

// list\_collection.add(2);

}

## Curiously Recurring Template Pattern (CRTP)

CRTP is a pattern where a class Derived inherits from a base class template that is instantiated with Derived itself (class Derived : public Base<Derived>).

**Example 1 – Static polymorphism**:

It allows you to add functionality to a class hierarchy **without the overhead of virtual functions**.

#include <iostream>

// Base class provides the interface

template <typename Derived>

class Counter {

public:

int count = 0;

void increment() {

// Downcast to the actual derived type to call its implementation

static\_cast<Derived\*>(this)->increment\_impl();

}

};

// Derived class 1

class SimpleCounter : public Counter<SimpleCounter> {

public:

void increment\_impl() {

count++;

std::cout << "SimpleCounter: " << count << std::endl;

}

};

// Derived class 2

class DoubleCounter : public Counter<DoubleCounter> {

public:

void increment\_impl() {

count += 2;

std::cout << "DoubleCounter: " << count << std::endl;

}

};

// This function works with any class that uses the Counter CRTP

template <typename T>

void useCounter(Counter<T>& c) {

c.increment();

c.increment();

}

int main() {

SimpleCounter sc;

DoubleCounter dc;

useCounter(sc);

useCounter(dc);

}

## Variadic Templates

Variadic templates can **take a variable number of template arguments**. This is the mechanism behind features like std::tuple, std::function, and std::make\_unique.

What does "variadic" mean?

typename... Args is called a *template parameter pack*. It declares a placeholder (Args) that can represent zero or more template type arguments.

A template (either a class or a function) that is declared with a parameter pack is called a *variadic template*.

Note: In computer science, variadic means a function or operator that can accept a variable number of arguments.

**Example 1 –** A type-safe printf function.

#include <iostream>

// Base case for recursion: no arguments left to print.

void safe\_printf(const char\* s) {

while (\*s) {

if (\*s == '%' && \*(++s) != '%') {

throw std::runtime\_error("invalid format string: missing arguments");

}

std::cout << \*s++;

}

}

// Recursive variadic template function

template<typename T, typename... Args>

void safe\_printf(const char\* s, T value, Args... args) {

while (\*s) {

if (\*s == '%' && \*(++s) != '%') {

std::cout << value << " ";

// Recurse with the rest of the arguments

return safe\_printf(++s, args...);

}

std::cout << \*s++;

}

throw std::logic\_error("extra arguments provided to printf");

}

int main() {

// This prints "Hello C++ world! The number is 1 and 2"

safe\_printf("Hello % world! The number is % and %.\n", "C++", 1, 2);

}

## Fold Expressions (C++17)

C++17 introduced fold expressions, a much more concise way to apply a binary operator to all elements in a parameter pack.

**Example 1 –** Summing all arguments.

#include <iostream>

template<typename... Args>

auto sum(Args... args) {

// Folds the '+' operator over the parameter pack 'args'

return (args + ...);

}

int main() {

std::cout << "Sum: " << sum(1, 2, 3, 4.5, 5) << std::endl; // 15.5

}

## SFINAE (Substitution Failure Is Not An Error)

This is a core principle that enables [*template metaprogramming*](#_Template_Metaprogramming_(TMP)). If a substitution of template parameters into a function signature results in an invalid construct, the compiler discards that overload instead of producing an error. It's often used with std::enable\_if.

**Use Case**: Creating a function that only works for integral types.

#include <iostream>

#include <type\_traits> // for std::enable\_if and std::is\_integral

// This overload is only enabled if T is an integral type.

template <typename T>

typename std::enable\_if<std::is\_integral<T>::value, void>::type

process(T value) {

std::cout << "Processing an integral type: " << value << std::endl;

}

// This overload is only enabled if T is NOT an integral type.

template <typename T>

typename std::enable\_if<!std::is\_integral<T>::value, void>::type

process(T value) {

std::cout << "Processing a non-integral type." << std::endl;

}

int main() {

process(100); // Calls the integral version

process(10.5); // Calls the non-integral version

}

## Concepts (C++20)

Concepts are a modern, more readable replacement for SFINAE. They allow you to specify constraints on template parameters directly.

**Example 1 –** Rewriting the SFINAE example with Concepts.

#include <iostream>

#include <concepts> // for std::integral

// Constrain the template parameter T to be an integral type.

template <std::integral T>

void process\_v2(T value) {

std::cout << "Processing an integral type: " << value << std::endl;

}

// A different function for other types (could be another concept or unconstrained)

template <typename T>

void process\_v2(T value) requires (!std::integral<T>) {

std::cout << "Processing a non-integral type." << std::endl;

}

int main() {

process\_v2(200);

process\_v2(20.5);

}

## if constexpr (C++17)

This allows you to have **compile-time branches** inside a function. The if is evaluated at compile time, and the code in the branch that is not taken is discarded and not compiled.

**Example 1 – A generic function that needs to do something special for pointer types:**

#include <iostream>

#include <type\_traits>

template <typename T>

void print\_value(T value) {

if constexpr (std::is\_pointer<T>::value) {

// This code only compiles if T is a pointer

std::cout << "Pointer value: " << \*value << std::endl;

} else {

// This code only compiles if T is not a pointer

std::cout << "Value: " << value << std::endl;

}

}

int main() {

int x = 10;

print\_value(x);

print\_value(&x);

}

# Template Metaprogramming (TMP)

TMP is a programming technique where you **use C++ templates to execute code at compile time**.

Instead of writing a program that runs when a user executes it, you are writing a "program within a program" that the compiler runs during the compilation process. The output of this compile-time program is C++ code that then gets compiled into the final executable.

## Core Idea

Think of the C++ compiler as a simple programming language interpreter where:

* **Variables** are template parameters (e.g., typename T, int N).
* **Functions** are class templates.
* **Branching (if/else)** is handled by template specialization or if constexpr.
* **Loops** are achieved through recursion (a template instantiating itself with different parameters).

The result of a TMP "computation" can be a value, a type, or generated code.

## Examples

### Compile-Time Factorial

This is the "Hello, World" of TMP. We can write a template that calculates the factorial of a number. The compiler will compute the result and embed it as a constant in the final program.

#include <iostream>

// The "recursive step": Factorial<N> is N \* Factorial<N-1>

template <int N>

struct Factorial {

// The result is computed at compile time

static const int value = N \* Factorial<N - 1>::value;

};

// The "base case" for the recursion, using full specialization

template <>

struct Factorial<0> {

static const int value = 1;

};

int main() {

// The compiler calculates Factorial<5>::value during compilation.

// It expands Factorial<5> -> 5 \* Factorial<4> -> 5 \* 4 \* Factorial<3> ...

// until it hits Factorial<0>.

// The final code is equivalent to: const int result = 120;

const int result = Factorial<5>::value;

std::cout << "5! is " << result << std::endl;

// This would cause a compile error because the recursion would never end

// or hit a negative number, depending on compiler limits.

// int invalid = Factorial<-1>::value;

}

## Why Template Metaprogramming?

**Performance Optimization**: Computations that can be done at compile time have zero runtime cost. The factorial example is trivial, but this can be used for complex math, lookup table generation, etc.

**Static Assertions and Type Checking**: TMP is the engine behind static\_assert and type traits (<type\_traits>). You can write code that checks properties of types and fails to compile if certain conditions aren't met, leading to safer code.

template <typename T>

void process(T data) {

// This check happens at compile time. The program won't even be built

// if you try to call process() with a non-copyable type.

static\_assert(std::is\_copy\_constructible<T>::value, "Type must be copyable");

// ...

}

**Code Generation**: It can be used to generate repetitive code automatically, reducing boilerplate and potential for human error. This is how std::tuple and std::variant are implemented.

## Modern Template Metaprogramming

While the Factorial example shows the classic style, modern C++ has made TMP much easier and more readable:

* **constexpr**: Allows you to write normal-looking functions that the compiler can execute at compile time. This is often a much cleaner way to compute values.
* **Type Traits**: The <type\_traits> header provides a huge library of pre-built TMP "functions" to query information about types (e.g., std::is\_pointer, std::is\_integral).
* **if constexpr (C++17)**: Provides simple, readable compile-time branching inside a function.
* **Concepts (C++20)**: Provide a clean, powerful syntax for expressing constraints on template parameters, which is a primary use case for TMP.

# Type Inference (C++11)

Type inference refers to the **automatic detection of the data type** of an expression in a programming language.

C++11 introduces several new handy type inference capabilities that mean you can spend less time having to write out things the compiler already knows and focus on the logic.

## Type Deduction

### For Variables

If the compiler can infer the type of a variable at the point of declaration, you can just write auto instead of putting in the variable type. Make sure the variable is initialized with a specific value so that the compiler can know which type it is.

For example:

auto x = 4; // instead of int x = 4;

or

auto y = 10000000L; // instead of long y = 10000000;

This, of course, is not the intended use of auto. But it really shines when working with iterator and template as follows:

**Iterator**

std::vector<int> vec;

auto it = vec.iterator(); // instead of std::vector<int>::iterator it;

or

auto it = vec.begin(); // instead of std::vector<int>::iterator it = vec.begin();

**Template**

template <typename Base, typename Derive>

void makeAndProcessObject(const Derive& builder) {

Base val = builder.makeObject();

// do stuff with val

}

In this code, the Base object cannot be deduced by the template parameter. Every call must look like this:

Husky builder;

makeAndProcessObject<Dog>(builder);

Using auto reduces this ugliness because it knows the base type based on the derived type:

template <typename Derive>

void makeAndProcessObject(const Derive& builder) {

auto val = builder.makeObject();

// do stuff with val

}

Now you only need a single template parameter:

Husky builder;

makeAndProcessObject(builder);

**Q&A: Can we use *auto* for reference, pointer and const?**

Yes!

If you have a reference, auto will automaticaly pick up a value. But you can specify & to force it pick up a reference:

int& foo();

auto bar = foo(); // int

auto& baz = foo(); // int&

If you have a pointer, auto will automatically pick up pointerness:

int\* foo();

auto p\_bar = foo(); // int\*

But you can also (thankfully) be explicit about it, and indicate that the variable is a pointer:

int\* foo();

auto\* p\_baz = foo(); // int\*

And for const auto:

int& foo();

const auto& baz = foo(); // const int&

Or with pointers:

int\* foo();

const auto\* p\_bar = foo(); // const int\*

### For Lambda Expressions (C++14)

C++11 allows automatically deducing the return type (actually ignore the -> return-type part) of [lambda expressions](#_ihv636) whose body contains only a single return statement.

In C++14, this has been expanded in two ways.

**1.** It now works with more complex functions whose body contains **more than one return statement** as long as all return statements return the same type:

// Deduce return type of multiple return statements

[](int x) {

if(isX2) {

return x \* 2;

}

return x;

};

**2.** It now works with **all functions**, not just lambdas:

// Deduce return type of conventional functions

auto getLength() {

return foo.mLength;

}

// Deduce return type of function templates

template <typename T>

auto& f(T& t) {

return t;

}

There several other good reasons why deducing the return type is a plus:

* When you need to return a fairly **complex type** (such as an iterator) when searching into a STL container. The auto return type makes the function easier to write and to read.
* Using an auto return type enhances your ability to refactor. Take the getLength() in the above code as an example. If you don’t use auto, making a change to the type of mLength would **cause a series of cascading changes** in getLength() and other functions calling getLength(). But if we use auto for the return type, the compiler will silently make the change for us.

## Trailing Return Type

In all prior versions of C++ 11, the return value of a function had to go before the function:

int multiply(int x, int y);

In C++11, you can do that:

auto multiply(int x, int y) -> int;

So would you want to do this? Let's look at a simple example where it helps us: a class with an enum declared inside it:

class Person {

public:

enum PersonType {ADULT, CHILD, SENIOR};

void setPersonType(PersonType inPersonType);

PersonType getPersonType();

private:

PersonType m\_PersonType;

};

Here we have a simple class Person that has a type: whether the person is an adult, a child, or a senior citizen. Not much special about it, but what happens when you go to define the methods?

The setter is trivial to declare. You can use the enum type PersonType without any trouble:

void Person::setPersonType(PersonType inPersonType) {

m\_PersonType = inPersonType;

}

But the getter is a bit of a mess:

// ERROR: The compiler doesn't know what PersonType is because it is being used outside

// of the Person class

PersonType Person::getPersonType() {

return m\_PersonType;

}

You have to write:

Person::PersonType Person::getPersonType() {

return m\_PersonType;

}

to make the return value work correctly. This isn't a big deal, but it's pretty easy to do by mistake, and it can get much messier when *templates* are involved.

This is where the new return value syntax comes in. Because the return value goes at the end of the function, instead of before it, you don't need to add the class scope. By the point the compiler reaches the return value, it already knows the function is part of the Person class, so it knows what PersonType is.

auto Person::getPersonType() -> PersonType {

return m\_PersonType;

}

## *decltype*

While auto lets you declare a variable with a particular type, decltype lets you extract the type from a variable (or any expression).

**Example 1 – What is it:**

int x = 3;

decltype(x) y = x; // same as auto y = x;

**Example 2 – Combining with typedef:**

std::vector<int> vec;

typedef decltype(vec.begin()) IT; // same as typedef std::vector<int>::iterator IT;

IT another\_iterator;

Of couse, these are **not useful** uses of decltpe. In fact, in terms of readability, being explicit with something like auto or std::vector::iterator is way clearer.

**Example 3 – Using in template function** (useful in C++11 or lower)**:**

decltype *can* shine in situations where you want your code to adapt to whatever type comes out of an expression — especially with *templates* or *generic* *programming*. It can help avoid hardcoding types and make code more robust across changes.

template <typename T, typename U>

auto add(T a, U b) -> decltype(a + b) {

return a + b;

}

...

auto x = add(3, 4.5); // int + double → x is a double

auto y = add(10, 20); // int + int → y is an int

Without decltype, you'd have to guess or hardcode a common type, which isn't very flexible. But with it, you get type-safe, reusable, and scalable code.

Note: You can completely ignore the part in "-> decltype(a + b)" since C++14.

**Example 4 – Using in lambda function:**

For this code, you don't need decltype, just auto is enough:

#include <iostream>

int main() {

    auto lambda = [](int a, int b) { return a \* b + 0.5; };

    auto result = lambda(4, 5);

    std::cout << "Result: " << result << std::endl;  // Output: Result: 20.5

    return 0;

}

But in that case, auto won't work because it requires immediate initialization. So decltype saves the day when you need precisetypededuction *without* immediate value assignment:

#include <iostream>

void func(bool condition) {

    auto lambda = [](int a, int b) { return a \* b + 0.5; };

    // This code figures out the return type at compile time and makes result exactly that type — so you're guaranteed the right match

    decltype(lambda(0, 0)) result; // You declare result here

    if (condition) {

        result = lambda(4, 5); // You assign result here

    } else {

        result = lambda(1, 2);

    }

    std::cout << "Result: " << result << std::endl;  // Output: Result: 20.5

}

int main() {

    func(true);

    return 0;

}

Let’s upgrade our code by making func a non-void function:

#include <iostream>

decltype(auto) func(bool condition) { // Only work in C++14

    auto lambda = [](int a, int b) { return a \* b + 0.5; };

    // This code figures out the return type at compile time and makes result exactly that type — so you're guaranteed the right match

    decltype(lambda(0, 0)) result; // You declare result here

    if (condition) {

        result = lambda(4, 5); // You assign result here

    } else {

        result = lambda(1, 2);

    }

    return result;

}

int main() {

    auto result = func(true);

std::cout << "Result: " << result << std::endl;  // Output: Result: 20.5

    return 0;

}

decltype(auto) will tell the compiler: "Just deduce exactly what type this function returns based on the return statement."

# Range-Based Loops (C++11)

Range-based loop is a much more convenient way to iterate all elements of **any STL container** in C++11. With it, we no longer have to care about lengthy syntax.

## Syntax

for (range\_declaration : range\_expression) {

loop\_statement

};

## Examples

**1. Only iterate**

#include <iostream>

#include <vector>

#include <map>

int main() {

// Iterating over array

int a[] = {0, 1, 2, 3, 4, 5};

for (int n : a) {

std::cout << n << ' '; // Get element without an index

}

std::cout << '\n';

// Iterate over vector

std::vector<int> vec = {0, 1, 2, 3, 4, 5}; // uniform initialization in C++11

for (auto i : vec) { // can declare "int" instead of "auto"

std::cout << i << ' ';

}

std::cout << '\n';

// Printing string characters

std::string str = "Geeks";

for (char c : str) {

std::cout << c << ' ';

}

std::cout << '\n';

// Printing keys and values of a map

std::map<int, int> m({{1, 1}, {2, 2}, {3, 3}});

for (auto i : m) {

std::cout << '{' << i.first << ", " << i.second << "} ";

}

}

Output:

0 1 2 3 4 5

0 1 2 3 4 5

G e e k s

{1, 1} {2, 2} {3, 3}

**2. Iterate and modify**

#include <iostream>

#include <string>

int main() {

std::string str = "abC";

for (char &ch : str) { // Without the &, output will be "abC"

if (ch == 'C') {

ch = 'c';

}

}

std::cout << str;

}

Output:

abc

## Limitations

There are some limitations to note when using range-based loop:

* There is **no index** and **no iterator**. If your loop operation requires an index or an iterator, you’re better off using the standard for loop.
* The loop **always iterates all elements** in a container (from begin() to the element just before end() – unless you break out early). If you want to start somewhere other than begin() or stop somewhere ahead of time other than before end(), you’re better off using the standard for loop.
* The loop iterates **only in a forward direction**. If you want to iterate backward, you would need to use the standard for loop format with rbegin() and rend() iterators.

## Allow Our Own Data Structures to Use Range-Based Loop

Strings, arrays, and all STL containers can be iterated over with the new range-based for loop already. But what if you want to allow your own data structures to use the new syntax? Check this [post](https://www.cprogramming.com/c++11/c++11-ranged-for-loop.html).

# Smart Pointers (C++11)

## Advatages

What make smart pointers differ from raw pointers?

**No Explicit Delete**

With raw pointers, we have to free memory regions they allocated:

MyClass \*ptr = new MyClass();

ptr->doSomething();

// We must write this line to avoid memory leak

delete ptr;

However, with smart pointers, we **don’t need to explicitly call delete**.

Smart pointer is a **wrapper class** over a pointer with operator like \* and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a raw pointer can’t, such as automatic destruction (yes, we don’t have to explicitly use delete), reference counting and more.

The smart pointer’s destructor contains the call to delete. And because the smart pointer is declared on the **stack**, its destructor is invoked when it goes out of scope, even if an exception is thrown somewhere further up the stack. As a result, the dynamically-allocated memory in the **heap** will be automatically deleted (or reference count can be decremented).

The below code show you how to implement a simple smart pointer:

#include <iostream>

using namespace std;

class SmartPtr {

private:

    int\* \_ptr;

public:

    explicit SmartPtr(int\* ptr = NULL) {

        \_ptr = ptr;

    }

    ~SmartPtr() {

        delete \_ptr;

    }

    int& operator\*() {

        return \*\_ptr;

    }

};

int main() {

    SmartPtr ptr(new int());

    \*ptr = 20;

    cout << \*ptr;

    return 0;

}

**Ownership Management**

Another subtle problem lies in ownership. A third-party function returns a pointer which is dynamically-allocated data. So, who is responsible for the cleanup? There is no way to infer such information by simply looking at the return type.

In case a third-party function returns a smart pointer, you can quickly deduce its type, what you can do with it and how the data lifetime is managed.

**Exception Safety**

Raw pointer are not safe from exceptions. Let us look at the following example:

void AMethod() {

ClassA \*a = new ClassA;

AnotherMethod(); // It can throw an exception

delete a;

}

If an exception is thrown, the object a is never deleted.

The following example shows a safer and shorter way to do that. It uses auto\_ptr which is deprecated in C++11, but the old standard is still widely used.

void AMethod() {

std::auto\_ptr<ClassA> a(new ClassA);

AnotherMethod(); // It can throw an exception

}

No matter what happens, after creating the a object, it will be deleted as soon as the program execution exits from the scope.

**Multithread Safety**

Raw pointers are not safe in multithread. For example:

Thread 1:

Connection& connection = connections.GetConnection(connectionID);

connection.send(data);

// ...

Thread 2:

connections.DeleteConnection(connectionID);

// …

If both threads used the same connection ID, this will result in undefined behavior. Access violation errors are often very hard to find.

In these cases, when more than one thread accesses the same resource, it is very risky to keep pointers or references to the resources, because some other threads can delete it. It is **much safer to use smart pointers with reference counting, like shared\_ptr**. They use atomic operations for increasing/decreasing a reference counter, so it is thread safe.

## Types

### std::auto\_ptr

DEPRECATED. So never use it!

**Why?**

**Copying of auto\_ptr object will transfer an ownership from one object to another**. For example:

auto\_ptr<ClassA> a(new ClassA);     // deprecated, please check the text

auto\_ptr<ClassA> b = a;

a->SomeMethod();                 // will result in access violation error

… will result in an access violation error. Only object b will contain a pointer to the object of ClassA, while a will be empty. Trying to access a class member of the object a will result in an access violation error.

Here is the thumb of rules:

* Never use auto\_ptr inside STL containers. Copying of containers will leave source containers with invalid data. Some STL algorithms can also lead to invalidation of auto\_ptr.
* Never use auto\_ptr as a function argument since this will lead to copying, and leave the value passed to the argument invalid after the function call.
* If auto\_ptr is used for data members of a class, be sure to make a proper copy inside a copy constructor and an assignment operator, or disallow these operations by making them private.
* BEST WAY: Never use auto\_ptr.

### std::unique\_ptr

A unique\_ptr is **the ONLY owner of the object** it points to and no other smart pointers can point to it. Object can be moved to a new owner, but cannot been copied or shared. This prevents the pointer from being incorrectly deleted multiple times.

The size of a unique\_ptr is one pointer and it supports rvalue references for fast insertion and retrieval from C++ Standard Library collections.

#### Syntax

Defined in header <memory>

|  |  |
| --- | --- |
| template<  class T,  class Deleter = std::default\_delete<T>  > class unique\_ptr; | (1) (since C++11) |
| template <  class T,  class Deleter  > class unique\_ptr<T[], Deleter>; | (2) (since C++11) |

#### Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Parameter** | **Description** | **Reference** |
| get |  | Returns a pointer to the managed object. | [cpp](https://en.cppreference.com/w/cpp/memory/unique_ptr/get.html) |
| operator bool |  | Checks if the stored pointer is not null | [cpp](https://en.cppreference.com/w/cpp/memory/unique_ptr/operator_bool.html) |
| release |  | Returns a pointer to the managed object and releases the ownership of the managed object.  *get() returns nullptr after the call.*  *The caller is responsible for cleaning up the object (e.g. by use of get\_deleter()).* | [cpp](https://en.cppreference.com/w/cpp/memory/unique_ptr/release.html) |
| reset | ptr = pointer() | Replaces the managed object with an object pointed to by ptr. | [cpp](https://en.cppreference.com/w/cpp/memory/unique_ptr/reset.html) |
| get\_deleter |  | Returns the deleter object which would be used for destruction of the managed object. | [cpp](https://en.cppreference.com/w/cpp/memory/unique_ptr/get_deleter.html) |
| operator\*  operator-> |  | Dereferences pointer to the managed object. | [cpp](https://en.cppreference.com/w/cpp/memory/unique_ptr/operator%252A.html) |
| operator[] |  | Provides indexed access to the managed array. | [cpp](https://en.cppreference.com/w/cpp/memory/unique_ptr/operator_at.html) |
| swap | unique\_ptr& other | Swaps the managed objects and associated deleters of \*this and another unique\_ptr object other. | [cpp](https://en.cppreference.com/w/cpp/memory/unique_ptr/swap.html) |

#### Examples

##### Example 1 – Define a unique pointer

You can define aunique\_ptr that points to any object, like this:

std::unique\_ptr<int> p1(new int);

std::unique\_ptr<int[]> p2(new int[50]);

std::unique\_ptr<Object> p3(new Object("Lamp"));

Since C++14, it's also possible to construct a unique\_ptr with the help of the special function std::make\_unique, like this:

std::unique\_ptr<int> p1 = std::make\_unique<int>();

std::unique\_ptr<int[]> p2 = std::make\_unique<int[]>(50);

std::unique\_ptr<Object> p3 = std::make\_unique<Object>("Lamp");

If you can, always prefer to allocate objects using std::make\_unique. For [why](#_32hioqz).

##### Example 2 – Can't have multiple references to a unique pointer's data

#include <iostream>

#include <memory>

void compute(std::unique\_ptr<int[]> p) { ... }

int main() {

std::unique\_ptr<int[]> ptr1 = std::make\_unique<int[]>(1024);

std::unique\_ptr<int[]> ptr2 = ptr1; // ERROR!

// Copy is not allowed

compute(ptr1); // ERROR!

// 'ptr1' is passed by copy, and copy is not allowed

return 0;

}

Technically this happens because a unique\_ptr doesn't have a *copy constructor* and a *copy assignment operator*.

##### Example 3 – Move object owned by a unique pointer

#include <iostream>

#include <memory>

struct A {

    void printA() {

        std::cout << "A struct...." << std::endl;

    }

};

int main() {

    std::unique\_ptr<A> p1(new A);

    p1->printA();

    // displays address of the containing pointer

    std::cout << p1.get() << std::endl;

    // now address stored in p1 should get copied to p2

    std::unique\_ptr<A> p2 = std::move(p1);

    p2->printA();

    std::cout << p1.get() << std::endl;

    std::cout << p2.get();

    return 0;

}

Output:

A struct....

0x3514c20

A struct....

0

0x3514c20

**Note**: Only non-const unique\_ptr can transfer the ownership of the managed object to another unique\_ptr. If an object's lifetime is managed by a const unique\_ptr, it is limited to the scope in which the pointer was created.

##### Example 4 – Using array in unique pointer

#include <iostream>

#include <memory>

int main() {

  std::unique\_ptr<int[]> arrPtr(new int[5]);

  for (int i = 0; i < 5; ++i) {

    arrPtr[i] = i;

  }

  for (int i = 0; i < 5; ++i) {

    std::cout << arrPtr[i] << " ";

  }

  std::cout << std::endl;

  return 0;

}

Note:

* If our unique pointer is std::unique\_ptr<T[]>, then the delete operator is called when the pointer exits from a scope.
* If our unique pointer is std::unique\_ptr<T[]>, then the delete[] operator is called when the pointer exits from a scope.

##### Example 5 – Using a custom deleter

Using inline lambda function:

#include <iostream>

#include <memory>

#include <cstdio>

int main() {

    // Open file

    FILE\* rawFile = fopen("data.txt", "r");

    if (!rawFile) {

        std::cerr << "Failed to open file.\n";

        return -1;

    }

    // Wrap FILE\* in unique\_ptr with custom deleter

    std::unique\_ptr<FILE> filePtr(rawFile, [](FILE\* f) {

        if (f) {

            std::cout << "Closing file...\n";

            fclose(f);

        }

    });

    // Read and print one line

    char buffer[256];

    if (fgets(buffer, sizeof(buffer), filePtr.get())) {

        std::cout << "Read line: " << buffer;

    } else {

        std::cout << "Failed to read from file.\n";

    }

    // filePtr goes out of scope here, fclose is called automatically

    return 0;

}

Using lambda function with deduced type:

#include <iostream>

#include <memory>

#include <cstdio>

int main() {

    FILE\* rawFile = fopen("data.txt", "r");

    if (!rawFile) {

        std::cerr << "Failed to open file.\n";

        return 1;

    }

    auto fileDeleter = [](FILE\* f) {

        if (f) {

            std::cout << "Closing file...\n";

            fclose(f);

        }

    };

    std::unique\_ptr<FILE, decltype(fileDeleter)> filePtr(rawFile, fileDeleter);

    char buffer[256];

    if (fgets(buffer, sizeof(buffer), filePtr.get())) {

        std::cout << "Read line: " << buffer;

    }

    return 0;

}

Using lambda function with functor:

#include <iostream>

#include <memory>

#include <functional>

#include <cstdio>

int main() {

    FILE\* rawFile = fopen("data.txt", "r");

    if (!rawFile) {

        std::cerr << "Failed to open file.\n";

        return 1;

    }

    std::function<void(FILE\*)> fileDeleter = [](FILE\* f) {

        if (f) {

            std::cout << "Closing file...\n";

            fclose(f);

        }

    };

    std::unique\_ptr<FILE, std::function<void(FILE\*)>> filePtr(rawFile, fileDeleter);

    char buffer[256];

    if (fgets(buffer, sizeof(buffer), filePtr.get())) {

        std::cout << "Read line: " << buffer;

    }

    return 0;

}

### std::shared\_ptr

A shared\_ptr owns the object it points to, but also allows for **multiple owners**. An internal counter is decreased each time a shared\_ptr pointing to the same object goes out of scope. This technique is called **Reference Counting**. When the last reference is destroyed, the counter goes to 0 and the data will be deallocated. In other words, **the raw pointer is NOT deleted until all shared\_ptr owners have gone out of scope** **or given up ownership**.

This type of smart pointer is useful when you want to share your dynamically-allocated data around, the same way you would do with raw pointers or references.

The size of a shared\_ptr is two pointers; one for the object and one for the shared control block that contains the reference count.

#### Syntax

Defined in header <memory>

|  |  |
| --- | --- |
| template<  class T,  > class shared\_ptr; | (1) (since C++11) |

#### Interfaces

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Parameter** | **Description** | **Reference** |
| get |  | Returns a pointer to the managed object. | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/get.html) |
| operator bool |  | Checks if the stored pointer is not null | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/operator_bool.html) |
| reset | ptr = pointer() | Replaces the managed object with an object pointed to by ptr. | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/reset.html) |
| get\_deleter |  | Returns the deleter object which would be used for destruction of the managed object. | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/get_deleter.html) |
| operator\*  operator-> |  | Dereferences pointer to the managed object. | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/operator%252A.html) |
| operator[] |  | Provides indexed access to the managed array. | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/operator_at.html) (C++17) |
| swap | shared\_ptr& other | Swaps the managed objects and associated deleters of \*this and another shared\_ptr object other. | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/swap.html) |
| use\_count |  | Returns the number of shared\_ptr objects referring to the same managed object | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/use_count.html) |
| unique |  | Checks whether the managed object is managed only by the current shared\_ptr object | [cpp](https://en.cppreference.com/w/cpp/memory/shared_ptr/unique.html) (C++20) |

#### Examples

##### Example 1 – Define a shared pointer

You can define ashared\_ptr that points to any object, like this:

std::shared\_ptr<int> p1(new int);

std::shared\_ptr<Object> p2(new Object("Lamp"));

Since C++14, another way to build a shared\_ptr is using the special function std::make\_shared. For [why](#_32hioqz).

std::shared\_ptr<int> p1 = std::make\_shared<int>();

std::shared\_ptr<Object> p2 = std::make\_shared<Object>("Lamp");

##### Example 2 – Get number of references

One of the main features of shared\_ptr is the ability to track how many pointers refer to the same resource. You can get information on the number of references with the method use\_count(). Consider this:

void compute() {

std::shared\_ptr<int> ptr1 = std::make\_shared<int>(100); // ptr1.use\_count() == 1

std::shared\_ptr<int> ptr2 = ptr1; // ptr1.use\_count() == 2

// ptr2.use\_count() == 2

} // 'ptr1' and 'ptr2' go out of scope here. No more references to the

// object (use\_count() == 0), so it is automatically cleaned up.

##### Example 3 – Using array and custom deleter with shared pointer

A shared\_ptr is designed for managing single objects and does not have a std::unique\_ptr<T[]> specialization for arrays, like unique\_ptr.

So if you need to deallocate resouce managed by your shared\_ptr, you need **a *custom deleter***.

For example:

#include <iostream>

#include <memory>

struct ArrayDeleter {

  template <typename T>

  void operator()(T\* array) const {

    delete[] array;

  }

};

int main() {

  std::shared\_ptr<int> arrPtr(new int[5], ArrayDeleter());

  for (int i = 0; i < 5; ++i) {

    arrPtr.get()[i] = i;

  }

  for (int i = 0; i < 5; ++i) {

    std::cout << arrPtr.get()[i] << " ";

  }

  std::cout << std::endl;

  return 0;

}

Another example, but with lambda expression:

std::shared\_ptr<int[]> p2(new int[5], [] (int\* i) {

delete[] i; // Custom delete

});

Unfortunately there's no way to do this when using std::make\_shared.

#### Notes

##### Prevent Circular References

Using shared\_ptr **incorrectly can lead to memory leaks** because of Circular References. For example:

#include <iostream>

#include <memory>

struct Player {

std::shared\_ptr<Player> companion;

~Player() { std::cout << "Destructor\n"; } // Never get called

};

int main() {

std::shared\_ptr<Player> jasmine = std::make\_shared<Player>();

std::shared\_ptr<Player> albert = std::make\_shared<Player>();

jasmine->companion = albert; // (1)

albert->companion = jasmine; // (2)

}

We’re just created a circular reference. In (1), we give jasmine a pointer to albert-data, while in (2) albert holds a pointer to jasmine-data. This is like giving each player a companion.

When jasmine goes out of scope at the end of the program, its destructor never gets called because there is still one smart pointer pointing at jasmine-data; that is albert->companion. Likewise, when albert goes out of scope, its destructor never get called because a reference to albert-data still lives through jasmine->companion. At this point, the program **just quits without freeing memory**.

Luckily, you can **break the circular reference using one of following ways**:

// At the end of the main() function

jasmine->companion = nullptr;

albert->companion = nullptr;

Or:

// At the end of the main() function

jasmine->companion.reset();

albert->companion.reset();

##### Dealing with this pointer:

Passing the this pointer as a parameter to a function cannot be done directly. That’s because it is a raw pointer, while the smart pointer is an object.

class A : std::enable\_shared\_from\_this<A> {

public:

   std::shared\_ptr<B> createB() {

      return std::make\_shared<B>(std::shared\_from\_this());

      // return std::make\_shared<B>(this);                        // syntax error

   }

}

class B {

private:

   std::shared\_ptr<A> a;

public:

   B(std::shared\_ptr<A>);

}

Notes:

* + shared\_from\_this() should **never be called on an object that isn't already managed by a shared\_ptr**. This is valid:

shared\_ptr<A> a(new A); // "a" is owned by shared\_ptr

a->createB();

While the following causes undefined behaviour (C++17 throws std::bad\_alloc instead.):

A a; // "a" is NOT owned by shared\_ptr

a.createB();

* + Using shared\_from\_this() from a constructor is equivalent to using it on an object not owned by a shared\_ptr, because the objects is possessed by the shared\_ptr after the constructor returns.

### std::weak\_ptr

A weak\_ptr is basically a shared\_ptr, but it **doesn't increase the reference count**. It is useful when you want to observe an object, but do not require it to remain alive. It’s **extremely useful to break circular references** between shared\_ptr instances.

#### Syntax

Defined in header <memory>

|  |  |
| --- | --- |
| template<  class T,  > class weak\_ptr; | (1) (since C++11) |

#### Interfaces

* lock(): To safely access the underlying resource that the weak\_ptr is pointing to.

**Example:**

#include <iostream>

#include <memory>

int main() {

  std::shared\_ptr<int> sharedPtr = std::make\_shared<int>(42);

  std::weak\_ptr<int> weakPtr(sharedPtr);

  std::shared\_ptr<int> lockedPtr = weakPtr.lock();

  if (lockedPtr) {

    std::cout << "The value is: " << \*lockedPtr << std::endl;

  } else {

    std::cout << "The resource has been destroyed." << std::endl;

  }

  return 0;

}

Output:

The value is: 42

* expired(): to check whether the referenced object was already deleted.

#### Examples

##### Example 1 – Define a weak pointer

You can only create a weak\_ptr out of a shared\_ptr or another weak\_ptr. For example:

std::shared\_ptr<int> p\_shared = std::make\_shared<int>(100);

std::weak\_ptr<int> p\_weak1(p\_shared);

std::weak\_ptr<int> p\_weak2(p\_weak1);

In the example above p\_weak1 and p\_weak2 point to the same dynamic data owned by p\_shared, but the reference counter doesn't grow.

##### Example 2 – Break circular references

Now back to the example about jasmine and albert, we can **break circular references** using weak\_ptr:

#include <iostream>

#include <memory>

struct Player {

  std::weak\_ptr<Player> companion;

  ~Player() { std::cout << "Destructor\n"; }

};

int main() {

  std::shared\_ptr<Player> jasmine = std::make\_shared<Player>();

  std::shared\_ptr<Player> albert  = std::make\_shared<Player>();

  jasmine->companion = albert;  // (1)

  albert->companion = jasmine;  // (2)

  return 0;

}

## Q&A

### Are smart pointers slower than raw ones?

According to various sources ([here](https://stackoverflow.com/questions/22295665/how-much-is-the-overhead-of-smart-pointers-compared-to-normal-pointers-in-c) and [here](http://blog.davidecoppola.com/2016/10/performance-of-raw-pointers-vs-smart-pointers-in-cpp/)), **performance of smart pointers should be close to raw ones**. A little speed penalty might be present in std::shared\_ptr due to the internal reference counting. All in all, there is some overhead, but it shouldn't make the code slow unless you continuously create and destroy smart pointers.

### Why std::make\_unique() and std::make\_shared()?

These methods provide two advantages:

**1**. They let us **forget about the new keyword**. When working with smart pointers, we want to get rid of the new/delete combo, right?

**2**. They make your **code safe against exceptions**. Consider calling a function that takes two smart pointers in input, like this:

void function(std::unique\_ptr<A>(new A()), std::unique\_ptr<B>(new B())) { ... }

Suppose that new A() succeeds, but new B() throws an exception: you catch it to resume the normal execution of your program. Unfortunately, the C++ standard does not require that object A gets destroyed and its memory deallocated: memory silently leaks and there's no way to clean it up. By wrapping A and B into std::make\_unique() you are sure the leak will not occur:

void function(std::make\_unique<A>(), std::make\_unique<B>()) { ... }

The point here is that std::make\_unique<A>() and std::make\_unique<B>() are now temporary objects; and cleanup of temporary objects is simple: their destructors will be triggered and the memory freed.

### How to implement a simple unique\_ptr?

template <typename T> class UniquePtr

{

private:

    T\* \_ptr;

public:

    explicit UniquePtr(T\* ptr = nullptr) : \_ptr(ptr) {

    }

// The copy constructor and copy assignment operator are deleted

// to prevent accidental copying and sharing of ownership, enforcing the unique ownership semantics.

    UniquePtr(const UniquePtr&) = delete;

    UniquePtr& operator=(const UniquePtr&) = delete;

// The move constructor and move assignmen t operator

// transfer the ownership of the managed object from one UniquePtr to another.

    UniquePtr(UniquePtr&& other) noexcept : \_ptr(other.\_ptr) {

        other.\_ptr = nullptr;

    }

// The destructor is responsible for deallocating the memory pointed to by the pointer

    ~UniquePtr() {

        delete \_ptr;

    }

    UniquePtr& operator=(UniquePtr&& other) noexcept {

        if (this != &other) {

            delete \_ptr;

            \_ptr = other.\_ptr;

            other.\_ptr = nullptr;

        }

        return \*this;

    }

    T\* operator->() const {

        return \_ptr;

    }

    T& operator\*() const {

        return \*\_ptr;

    }

    T\* get() const {

        return \_ptr;

    }

    void reset(T\* ptr = nullptr) {

        delete \_ptr;

        \_ptr = ptr;

    }

};

### How to implement a simple shared\_ptr?

template <typename T>

class SharedPtr

{

private:

    T\* \_ptr;

    size\_t\* \_ref\_count;

public:

    explicit SharedPtr(T\* ptr = nullptr) : \_ptr(ptr), \_ref\_count(new size\_t(1)) {}

    // The copy constructor and copy assignment operator

    // increase the reference count and share ownership of the managed object.

    SharedPtr(const SharedPtr& other) : \_ptr(other.\_ptr), \_ref\_count(other.\_ref\_count) {

        ++(\*\_ref\_count);

    }

    // The destructor decreases the reference count and deallocates the memory pointed to by the pointer

    // and the reference count object if the count reaches 0.

    ~SharedPtr() {

        if (--(\*\_ref\_count) == 0) {

            delete \_ptr;

            delete \_ref\_count;

        }

    }

    SharedPtr& operator=(const SharedPtr& other) {

        if (this != &other) {

            if (--(\*\_ref\_count) == 0) {

                delete \_ptr;

                delete \_ref\_count;

            }

            \_ptr = other.\_ptr;

            \_ref\_count = other.\_ref\_count;

            ++(\*\_ref\_count);

        }

        return \*this;

    }

    T\* operator->() const {

        return \_ptr;

    }

    T& operator\*() const {

        return \*\_ptr;

    }

    T\* get() const {

        return \_ptr;

    }

    // Deallocates the current object and takes a new raw pointer as an argument,

    // allowing the shared pointer to manage a new object.

    void reset(T\* ptr = nullptr) {

        if (--(\*\_ref\_count) == 0) {

            delete \_ptr;

            delete \_ref\_count;

        }

        \_ptr = ptr;

        \_ref\_count = new size\_t(1);

    }

};

# Move Semantics (C++11)

## Rvalues and Lvalues

An **lvalue is an expression whose address can be taken.** It has a permanent or semi-permanent piece of memory.

Examples:

// 1. It can be a variable:

int a;

a = 1; // 'a' is an lvalue

// 2. Or not a variable:

int x;

int& getRef() {

    return x;

}

getRef() = 4;   // 'getRef' returns a reference to a global variable, so it's an lvalue

// 3:

int x = 10;

int& ref = x; // 'ref' is an lvalue reference to 'x'

// 4:

int arr[5]; // 'arr' is an lvalue

On the other hand, **rvalue is an expression that results in a temporary memory**.

// 1:

int result = 2 + 3; // '2 + 3' is an rvalue

// 2:

int x;

int getVal() {

    return x;

}

getVal();   // the return of getVal() is a temporary value

// 3.

int&& rref = 2 + 3; // 'rref' is an rvalue reference

Move semantics, specifically move operations like move constructors and move assignment operators, take advantage of rvalues to efficiently transfer the resources (such as dynamically allocated memory) from one object to another, without the need for expensive deep copying.

**Note**: Rvalue references are denoted with a &&.

## Move Constructor

**What:**

A move constructor is used to constructs an object with an object of the same type by "moving" the resources owned by the existing object.

**Why:**

Move constructors are used for following reasons:

* Unlike copy constructors, which create a new object by copying the data of an existing object, **move constructors transfer ownership of resources by updating the internal pointers of the existing object to point to the resources of the *temporary object* being moved from**. Thus, move constructor **prevents unnecessarily copying data in the memory**.

*What is "temporary object"?*

* It refers to an **unnamed object** created implicitly or explicitly in certain situations, like during function calls or when using rvalue expressions. In the context of move semantics, it typically refers to an **rvalue**. Rvalues can include temporary objects, literals, or the result of an expression.

*But temporary objects reside in temporary memory regions and are deallocated when out of scope, so should the new object share the same pointer with the temporary object?*

* Correct. That’s why the move constructor or move assignment operator must transfer ownership of resources from the temporary object to the destination object. After the transfer is complete, whether the temporary object is needed or not doesn’t matter.

**Syntax:**

The general form of a move constructor is:

ObjectName(ObjectName&& source)

: data { source.data }

{

source.data = nullptr;

}

**Note**: In a move operation, the goal is to transfer ownership of resources from the source object to the destination object. After the move, the source object should be in a valid but unspecified state, ready to be safely destructed or assigned to. That’s why we should set source.data to nullptr.

**Examples:**

First, let’s see what the problem of copy constructors:

#include <iostream>

#include <vector>

using namespace std;

class Move {

private:

    int\* data;

public:

    Move(int d)

    {

        data = new int;

        \*data = d;

        cout << "Constructor is called for " << d << endl;

    };

    // Copy constructor

    Move(const Move& source)

: Move { \*source.data }

    {

        cout << "Copy Constructor is called for " << \*source.data << endl;

    }

    ~Move()

    {

        cout << "Destructor is called for " << \*data << endl;

        delete data;

    }

};

int main()

{

    vector<Move> vec;

    cout << "==" << endl;

    Move move(1);

    cout << "==" << endl;

    vec.push\_back(move);

    cout << "==" << endl;

    return 0;

}

Output:

==

Constructor is called for 1

==

Constructor is called for 1

Copy Constructor is called for 1

==

Destructor is called for 1

Destructor is called for 1

The above program shows unnecessarily calling copy constructor and inefficiently using the memory by copying the same data several times.

Now, let’s see how move constructor resolves the problem of copy constructor:

#include <iostream>

#include <vector>

using namespace std;

class Move {

private:

    int\* data;

public:

    Move(int d)

    {

        data = new int;

        \*data = d;

        cout << "Constructor is called for " << d << endl;

    };

    // Copy constructor

    Move(const Move& source)

: Move { \*source.data }

    {

        cout << "Copy Constructor is called for " << \*source.data << endl;

    }

    // Move Constructor

    Move(Move&& source)

: data { source.data }

    {

        cout << "Move Constructor for " << \*source.data << endl;

        source.data = nullptr;

    }

    ~Move()

    {

        if (data != nullptr) {

            cout << "Destructor is called for " << \*data << endl;

        }

        else {

            cout << "Destructor is called for nullptr" << endl;

        }

        delete data;

    }

};

int main()

{

    vector<Move> vec;

    cout << "==" << endl;

    vec.push\_back( Move(1) );

    cout << "==" << endl;

// Move constructor only works with rvalue,

// so doing this "Move move(1); vec.push\_back(move);" won’t call the move constructor

    return 0;

}

Output:

==

Constructor is called for 1

Move Constructor for 1

Destructor is called for nullptr

==

Destructor is called for 1

As seen, the unnecessary call to the copy constructor is avoided by making the call to the move constructor.

**The std::move Method:**

In the above example, we can change the main function like that:

int main()

{

    vector<Move> vec;

    Move move(1);

    vec.push\_back( std::move(move) );

    return 0;

}

## Move Assignment Operator

**What:**

A move assignment operator is used to assign an rvalue to an existing object of the same type by "moving" the resources owned by the rvalue.

**Why:**

Just like move constructors, move assignment operators help **prevent unnecessarily copying data in the memory**.

**Syntax:**

The general form of a move assignment operator is:

ObjectName& operator=(ObjectName&& source) {

if (this != &source) {

delete data;

// Move the resources from 'source' into 'this' object

source.data = nullptr;

}

return \*this;

}

**Examples**:

#include <iostream>

#include <vector>

using namespace std;

class Move {

private:

    int\* data;

public:

    Move(int d) {

        data = new int;

        \*data = d;

        cout << "Constructor is called for " << d << endl;

    };

    // Copy constructor

    Move(const Move& source) : Move { \*source.data }   {

        cout << "Copy Constructor is called for " << \*source.data << endl;

    }

    // Move Constructor

    Move(Move&& source) : data { source.data }  {

        cout << "Move Constructor for " << \*source.data << endl;

        source.data = nullptr;

    }

    // Move assignment operator

    Move& operator=(Move&& source) {

        if (this != &source) {

            delete data;

            data = source.data;

            source.data = nullptr;

            cout << "Move Assignment Operator for " << \*data << endl;

        }

        return \*this;

    }

    ~Move() {

        if (data != nullptr) {

            cout << "Destructor is called for " << \*data << endl;

        }

        else {

            cout << "Destructor is called for nullptr" << endl;

        }

        delete data;

    }

};

int main()

{

    Move move1(1);

    Move move2(2);

    move2 = std::move(move1);       // Now "move2" will be replaced by "move1"

    return 0;

}

Output:

Constructor is called for 1

Constructor is called for 2

Move Assignment Operator for 1

Destructor is called for 1

Destructor is called for nullptr

## When and When Not?

Use Move Semantics when:

* **Expensive-to-copy objects**: If objects have a large memory footprint or involve complex computations, it’s often more efficient to move them than to copy them.
* **Rvalue references**: Move semantics are designed to work with rvalue references (&&). So, when you have an rvalue, utilizing move semantics can result in more efficient resource management.

Avoid Move Semantics when:

* **Small objects**: The overhead of moving small objects may be comparable to or even higher than the overhead of copying them.
* **Shared ownership**: If objects are shared among multiple owners (e.g., using shared pointers), move semantics may not be suitable. Moving these objects would invalidate the other owners' references and potentially lead to undefined behavior.
* **Immutable objects**: If an object is immutable, move semantics are typically unnecessary. Because the object's state cannot change, making copies instead of moving has no significant impact on performance.

**Note**: **Modern C++ compilers and the standard library can eliminate unnecessary copies even without explicitly using move semantics**. That’s because they can apply optimizations automatically, like Return Value Optimization (RVO), Named Return Value Optimization (NRVO), Small Object Optimization (SOO), Copy Elision, etc. So, it's worth considering whether move semantics are genuinely needed in a given cases. It should be used when there is a clear benefit in terms of performance or resource management.

# Any (C++17)

## What Is std::any?

std::any, introduced in C++17, is a type-safe container for single values of any copy-constructible type. It provides a way to **hold a value of any type** and retrieve it later in a type-safe manner. It is defined in the <any> header.

Think of it as a wrapper that can hold an int, a std::string, a custom class object, or any other type, one at a time.

## Built-In Functions

|  |  |
| --- | --- |
| **Function** | **Description** |
| ctor | You can create an empty std::any, or initialize it with a value. |
| any\_cast | This is the primary function to retrieve the contained value. It will throw a std::bad\_any\_cast exception if you try to cast to the wrong type. You can also cast to a pointer type, which will return nullptr on a type mismatch instead of throwing. |
| has\_value() | Returns true if the std::any object contains a value, false otherwise. |
| reset() | Destroys the contained object, leaving the std::any empty. |
| type() | Returns the std::type\_info of the contained type. |
| emplace() | Constructs an object in-place within the std::any. |

## Use Cases

### Heterogeneous Collections

Storing objects of different types in a single container.

std::vector<std::any> my\_stuff;

my\_stuff.push\_back(42);

my\_stuff.push\_back(std::string("hello"));

my\_stuff.push\_back(12.34f);

### Configuration/Property Systems

Storing configuration values that can be of various types (e.g., int, bool, std::string).

std::map<std::string, std::any> config;

config["port"] = 8080;

config["hostname"] = std::string("localhost");

config["enable\_tls"] = true;

### Generic Message/Event Systems

Passing arbitrary data payloads with events or messages.

## Examples

Here is a complete example demonstrating the creation, storage, and safe retrieval of values using std::any.

#include <iostream>

#include <any>

#include <vector>

#include <string>

#include <typeinfo>

// A custom class to store in std::any

class MyClass {

public:

int value = 100;

};

void processAny(const std::any& a) {

if (!a.has\_value()) {

std::cout << "Empty any" << std::endl;

return;

}

std::cout << "Contains a value of type: " << a.type().name() << std::endl;

// Safe access using pointer cast (returns nullptr on failure)

if (const int\* i = std::any\_cast<int>(&a)) {

std::cout << "It's an int: " << \*i << std::endl;

} else if (const std::string\* s = std::any\_cast<std::string>(&a)) {

std::cout << "It's a string: \"" << \*s << "\"" << std::endl;

} else if (const MyClass\* mc = std::any\_cast<MyClass>(&a)) {

std::cout << "It's MyClass with value: " << mc->value << std::endl;

} else {

std::cout << "Unknown type." << std::endl;

}

}

int main() {

std::vector<std::any> items;

// Store different types

items.push\_back(42);

items.push\_back(std::string("Hello C++17"));

items.push\_back(MyClass());

items.push\_back({}); // An empty any

// Modify an item in place

items[3].emplace<float>(99.9f);

for (const auto& item : items) {

processAny(item);

}

std::cout << "\n--- Demonstrating bad\_any\_cast ---\n";

std::any val = 5;

try {

// This works

std::cout << "Casting to int: " << std::any\_cast<int>(val) << std::endl;

// This will throw

std::cout << "Casting to double..." << std::endl;

std::cout << std::any\_cast<double>(val) << std::endl;

} catch (const std::bad\_any\_cast& e) {

std::cerr << "Caught exception: " << e.what() << std::endl;

}

return 0;

}

# Noexcept

## What Is noexcept?

The noexcept keyword tells the compiler **whether a function is *guaranteed* not to throw exceptions**. There are two main uses:

* noexcept specifier: In function declarations or definitions. To declare that a function won’t throw. Like "I swear this function won’t throw."
* noexcept(expr) operator: Evaluates to true or false at compile time. Like "Can this expression throw?"

## Why Use noexcept?

* **Improves performance**: Compilers can skip generating exception-handling code (like stack unwinding) for noexcept functions.
* **Enables better optimizations**: STL containers and algorithms will favor *move constructors* or *operations* marked noexcept.
* **Improves reliability in templates**: Lets you write SFINAE-friendly or exception-aware generic code.
* **Enables compile-time validation**: When used with the noexcept(expr) operator and static\_assert.

**Note**: noexcept cannot help improve code security, like preventing crashes. It's created to focus on improving performance optimization, especially in STL-heavy or move-intensive code.

## Examples

### Basic noexcept Function

void logMessage() noexcept {

... // any operation, even one which could throw exception

}

Now the compiler sees a contradiction: you said "I swear this function won’t throw."

But if there is, the program will call std::terminate(), then immediate termination — no stack unwinding, no chance to catch anything.

### Using try/catch inside noexcept

All exceptions will be caught, just like usual.

But you promise to the compiler that "Don’t worry — I’ll never let an exception escape me."

void safe() noexcept {

try {

throw std::runtime\_error("Handled inside");

} catch (...) {

std::cout << "All good!" << std::endl;

}

}

But this would crash. You promise to the compiler but then you failed:

void boom() noexcept {

throw std::runtime\_error("Uncaught!");

}

Of course, if there is no noexcept here, your program will also crash (but with stack unwinding).

### noexcept(expr) Operator

This is a **compile-time trait** that returns true if the expression is noexcept. Mostly used with static\_assert:

void mightThrow();

void guaranteed() noexcept {}

static\_assert(!noexcept(mightThrow()), "Not safe!");

static\_assert( noexcept(guaranteed()), "Totally safe!");

### Conditional noexcept in Templates

template <typename T>

void callSomething(T t) noexcept(noexcept(t.doSomething())) {

t.doSomething();

}

The inner noexcept asks the compiler: *“Will t.doSomething() throw?”* If doSomething() is marked noexcept, the answer is true. Otherwise, false.

The outer noexcept becomes either noexcept(true) if the doSomething() is marked as noexcept or noexcept(false) if doSomething() is not marked noexcept.

This means the callSomething() inherits the noexcept-ness of doSomething().

### Move Semantics & STL Optimization

STL containers like std::vector **prefer move** if it’s noexcept. Otherwise, they fall back to copy to avoid leaving the container in an unstable state if an exception is thrown halfway through.

|  |  |
| --- | --- |
| #include <iostream>  #include <vector>    class MyClass {  public:  MyClass() {  std::cout << "Default constructor\n";  }    MyClass(const MyClass&) {  std::cout << "Copy constructor\n";  }    MyClass(MyClass&&) {  std::cout << "Move constructor\n";  }  };    int main() {  std::vector<MyClass> vec;    std::cout << "--- Pushing first object ---\n";  vec.push\_back(MyClass{}); // Should use move    std::cout << "--- Pushing second object ---\n";  vec.push\_back(MyClass{}); // May reallocate + move or copy    return 0;  } | --- Pushing first object ---  Default constructor  Move constructor  --- Pushing second object ---  Default constructor  Move constructor  Copy constructor |
| For the same above code, if we add noexcept to the move constructor, we'll have a different outout:  …  MyClass(MyClass&&) noexcept {  std::cout << "Move constructor\n";  }  … | --- Pushing first object ---  Default constructor  Move constructor  --- Pushing second object ---  Default constructor  Move constructor  Move constructor |

### Overload Resolution Based on noexcept

You can guide overload selection using the noexcept(expr) trait.

void executeImpl() noexcept { std::cout << "Noexcept version\n"; }

void executeImplThrowing() { std::cout << "Throwing version\n"; }

template <typename F>

void execute(F f) {

if constexpr (noexcept(f()))

executeImpl();

else

executeImplThrowing();

}

# Other New Features in C++11

## In-Class Initialization of Data Members

C++11 supports in-class initialization of data members:

class Foo {

int a = 0; // C++11 only

public:

Foo();

};

## Type Aliases

Semantically similar to using a typedef however, type aliases with using are easier to read and are compatible with templates.

template <typename T>

using Vec = std::vector<T>;

Vec<int> v; // std::vector<int>

using String = std::string;

String s {"foo"};

## std::for\_each

We alread knew about conventional for loop and [range-based loop](#_3o7alnk), it’s time to discover another kind of for loop (actually a built-in *function*) in C++11 – the std::for\_each() in header file <algorithm>.

### Syntax

for\_each(first, last, func)

* first : The beginning position from where operation has to be executed.
* last : The ending position utill where operation has to be executed.
* func : The function or function object or lambda expression which operation will be applied to each element.

### Example

#include <iostream>

#include <vector>

#include <algorithm>

void printNum(int a) {

std::cout << a << " ";

}

class Functor {

public:

void operator()(int a) {

std::cout << a << " ";

}

};

int main() {

int arr[5] = { 1, 2, 3, 4, 5 };

std::vector<int> vec = { 5, 4, 3, 2, 1 };

Functor functor;

std::cout << "Using array and function: ";

std::for\_each(arr, arr + 5, printNum);

std::cout << std::endl;

std::cout << "Using array and functor: ";

std::for\_each(arr, arr + 5, functor);

std::cout << std::endl << std::endl;

std::cout << "Using vector and function: ";

std::for\_each(vec.begin(), vec.end(), printNum);

std::cout << std::endl;

std::cout << "Using vector and functor: ";

std::for\_each(vec.begin(), vec.end(), functor);

// Using lambda ...

}

Output:

Using array and function: 1 2 3 4 5

Using array and functor: 1 2 3 4 5

Using vector and function: 5 4 3 2 1

Using vector and functor: 5 4 3 2 1

### Why std::for\_each() ?

* Similar to range-based loop, the std::for\_each can work with **any STL container**. But it’s much more versatile because it allows using **index or iterator** when needed.
* Comparing to conventional for loop, the std::for\_each is much **more** **readable**. It allows you to call a function, a functor or a lambda expression (so you can write an algorithm) on top of the loop.
* It opens your mind to the many STL-algorithms (eg, find\_if, sort, replace, etc.) which are functors. So using a std::for\_each, you can easily call these algorithms.

## Regex

<https://en.cppreference.com/w/cpp/regex>

# Other New Features in C++14

## Digit Separators

The single-quote character ' can now be used anywhere within a numeric literal for readability. It does not affect the numeric value.

auto million = 1'000'000;

auto pi = 3.14159'26535'89793;

## Binary Literals

C++ now supports binary literals:

auto a1 = 42; // decimal

auto a2 = 0x2A; // hexadecimal

auto a3 = 0b101010; // binary

auto a4 = 0b1111'1111 // = 255

## The [[deprecated]] Attribute

C++14 introduces the [[deprecated]] attribute to indicate that a unit (variable, function, class, etc) is discouraged. The compiler will, therefore, **output warning message**.

[[deprecated]]

void old\_method();

[[deprecated("Use new\_method instead")]]

void legacy\_method();

# Constant Expressions (constexpr)

## What Is constexpr?

**Definition**

A constexpr specifier on a variable or function tells the compiler: This entity, where required, must be **evaluated at compile time**.

**Key Goals**

* Catch logic errors early (invalid compile-time expressions become hard errors).
* Precompute values so your program **runs faster** and **uses less memory** at runtime. You will surprise that precompute expensive tables, factorials, string hashes, etc., now can be computed at compile time.
* Feed results into contexts that demand compile-time constants (array bounds, template arguments, static\_assert, etc.).

## constexpr vs. const

|  |  |  |
| --- | --- | --- |
| **Feature** | const | constexpr |
| Evaluation time | **Runtime** (unless optimizer folds it) | Always **compile time** |
| Function support | N/A | constexpr functions can be used |
| Template parameters | Cannot be a non-type parameter | Can be used as a non-type template argument (since C++11) |

## constexpr vs consteval

* constexpr: Can run at compile time or runtime (depending on context).
* consteval: Must run at compile time — or it’s a compilation error. Introduced in C++20.

Example:

#include <iostream>

consteval int doubleValue(int x) {

    return x \* 2;

}

int main() {

    int x = 5; // No error if "constexpr int x = 5;"

    int result = doubleValue(x);   // Error! x is not known at compile time

}

## History & Evolution

* **C++11**
  + constexpr variables: Must be initialized by a constant expression.
  + constexpr functions: Single return statement, no local variables/loops/branches.
* **C++14**
  + constexpr functions: You can have loops, local variables, multiple statements, if/switch. Return type can be deduced with auto in a constexpr function.
* **C++17**
  + constexpr lambdas are allowed.
  + Variables of literal class types with constexpr constructors can be declared at namespace scope.
* **C++20**
  + Even more freedom: dynamic allocation (new/delete), virtual calls in constexpr contexts, std::string operations become constexpr, etc.
  + consteval ("immediate" functions) guarantees compile-time only execution.

## Supported Data Types

In C++14 or lower, **only *literal types*** can be used in constexpr variables or functions.

A literal type must:

* Be a scalar (like int, float, bool, etc.)
* Be a class/struct type with:
  + A constexpr constructor.
  + All data members themselves of literal types.
  + Trivial destructors (i.e., no custom code inside ~MyType()).
  + No virtual base classes or virtual functions.

Since C++20, constexpr can be used with many standard types that were previously off-limits — like std::string, std::vector, and more.

## Constexpr vs Macro

Both constexpr and macros can result in **compile-time value substitution** in many contexts. But they have many differences:

|  |  |  |
| --- | --- | --- |
|  | Macro | Constexpr |
| Scope and Type Safety | #define VALUE 200U  // Literally replaces "VALUE" with "200U" in source code  // No type checking, no scope rules | static constexpr uint32\_t VALUE = 200U;  // Creates a typed constant with proper C++ scoping  // Type-safe, follows namespace/class rules |
| When Substitution Happens | Cases where constexpr behaves like macro (value substitution):  constexpr int SIZE = 10;    // Compile-time contexts:  int array[SIZE]; // ✅ Array size  static\_assert(SIZE == 10); // ✅ Static assertion  case SIZE: break; // ✅ Switch case  template<int N = SIZE> void f(); // ✅ Template parameter    // C-style variadic functions:  printf("%d", SIZE); // ✅ Value passed on stack  sprintf(buf, "%d", SIZE); // ✅ Value passed on stack  fprintf(file, "%d", SIZE); // ✅ Value passed on stack | Cases where constexpr does NOT behave like macro (symbol needed):  constexpr int SIZE = 10;    // Address-taking:  const int\* ptr = &SIZE; // ❌ Taking address  const int& ref = SIZE; // ❌ Reference binding    // C++ template functions with perfect forwarding:  template<typename... T>  void log\_template(const char\* fmt, T&&... args);  log\_template("Value: %d", SIZE); // ❌ Perfect forwarding needs address    // Passing by reference to regular functions:  void func(const int& param);  func(SIZE); // ❌ Reference parameter needs address |
| ODR (One Definition Rule) Impact | #define TIME\_RETRY\_MS 200U  printf("Time: %u", TIME\_RETRY\_MS);  // Becomes: printf("Time: %u", 200U);  → **Never** causes ODR-usage because it's just text | static constexpr uint32\_t TIME\_RETRY\_MS = 200U;  printf("Time: %u", TIME\_RETRY\_MS);  // Need actual variable if value is passed by reference  → **Can** cause ODR-usage depending on context |

## Examples

### constexpr Variables

// Integral types

constexpr int num = 42;

constexpr bool flag = (num % 2 == 0);

// Floating point

constexpr double pi = 3.141592653589793;

// User-defined literal type (C++11)

struct Point {

    double x, y;

    constexpr Point(double x\_, double y\_) : x(x\_), y(y\_) {}

};

constexpr Point origin{0.0, 0.0};

// Array with compile-time size

constexpr std::size\_t SIZE = 5;

int arr[SIZE];      // OK

// Char pointer is a bit tricky, but possible

constexpr const char\* greeting = "Hello, world!";

// constexpr char\* bad = "Oops!";  // ERROR: trying to assign to non-const pointer

// std::string should be used (C++20)

constexpr std::string greeting("Hello, world!");

### constexpr Functions

#### Simple Function (C++11)

// Must be a single return expression

constexpr int square(int x) {

    return x \* x;

}

static\_assert(square(5) == 25, "square() must work at compile time");

#### Complex Function (C++14)

// Factorial can be computed at compile time

constexpr int factorial(int n) {

    int result = 1;

    for (int i = 2; i <= n; ++i) {

        result \*= i;

    }

    return result;

}

constexpr int F6 = factorial(6);

#### auto Return & Template (C++14)

template <typename T>

constexpr auto add(T a, T b) {

    return a + b;

}

static\_assert(add(2, 3) == 5);

static\_assert(add(2.5, 3.1) > 5.5);

#### constexpr Lambdas (C++17)

constexpr auto mul = [](int a, int b) { return a \* b; };

static\_assert(mul(3, 4) == 12);

Even generic lambdas can be constexpr:

constexpr auto add = [](auto a, auto b) { return a + b; };

### constexpr Variables in Class

In C++, a class member declared constexpr must also be static, because:

* constexpr implies compile-time evaluation.
* But non-static data members belong to each individual object instance—and can only be initialized in constructors.
* Compile-time evaluation requires the value to exist independently of any instance, which is what static provides.

But using static constexpr requires extra care as it can cause linker error:

|  |  |
| --- | --- |
| WORK in C++17, but cannot work in C++14  Due to: *main.cpp:(.text+0x3a): undefined reference to 'MyClass::CONST1'*  #include <cstdio>    class MyClass {  public:  static constexpr int CONST1 = 10; // only declaration, no definition (no address)  };    void log(const int& input) { // passed by reference  printf("input: %d", input);  }    int main() {  log(MyClass::CONST1);  return 0;  }  Why this code can works in C++17? C++17 implicitly add inline to all variables marked with static constexpr. So they have both declaration and definition. | WORK in all C++:  #include <cstdio>    class MyClass {  public:  static const int CONST1;  };    const int MyClass::CONST1 = 42;    void log(const int& input) {  printf("input: %d", input);  }    int main() {  log(MyClass::CONST1);  return 0;  } |
| DOES NOT WORK in all C++  Due to: error: *non-static data member ‘CONST1’ declared ‘constexpr'*  #include <cstdio>    class MyClass {  public:  constexpr int CONST1 = 10;  };    void log(const int& input) {  printf("input: %d", input);  }    int main() {  MyClass myclass;  log(myclass.CONST1);  return 0;  } |  |

### constexpr Constructors & Classes (C++11)

// all at compile time

struct Complex {

    double re, im;

    constexpr Complex(double r, double i) : re(r), im(i) {}

    constexpr Complex operator+(Complex o) const {

        return {re + o.re, im + o.im};

    }

};

constexpr Complex z1{1.0, 2.0}, z2{3.0, 4.0};

constexpr Complex z3 = z1 + z2;

### if constexpr (C++17)

template <typename T>

void print\_if\_integral(T x) {

    if constexpr (std::is\_integral<T>::value) {

        std::cout << x << " is integral\n";

    } else {

        std::cout << x << " is not integral\n";

    }

}

print\_if\_integral(10);    // calls the integral branch

print\_if\_integral(3.14);  // calls the floating branch

### Dynamic memory (C++20)

constexpr std::vector<int> make\_vec() {

    std::vector<int> v;

    for (int i = 0; i < 5; ++i) {

        v.push\_back(i);

    }

    return v;

}

constexpr auto v = make\_vec();

## How constexpr Effects Compilation Time?

Using too many constexpr will increase the compilation time without a doubt.

But for operations that requires tons of recursion or loops, **compilation can even get error** due to *recursion depth limits* (for recursive constexpr) or *overflow* (for iterative constexpr). For example:

#include <iostream>

constexpr int factorial\_recursive(int n) {

    return (n <= 1) ? 1 : (n \* factorial\_recursive(n - 1));

}

constexpr int factorial\_nonrecursive(int n) {

    int result = 1;

    for (int i = 2; i <= n; ++i) result \*= i;

    return result;

}

int main() {

    constexpr int fact = factorial\_recursive(20);        // Compile error. Changing 20 to 10 resolves the error

    // constexpr int fact = factorial\_nonrecursive(20);  // Compile error. Changing 20 to 10 resolves the error

}

In general, do not overuse of constexpr**.** And **prefer iterative implementation over recursion** — it’s much easier on the compiler.

# Precompiled Headers

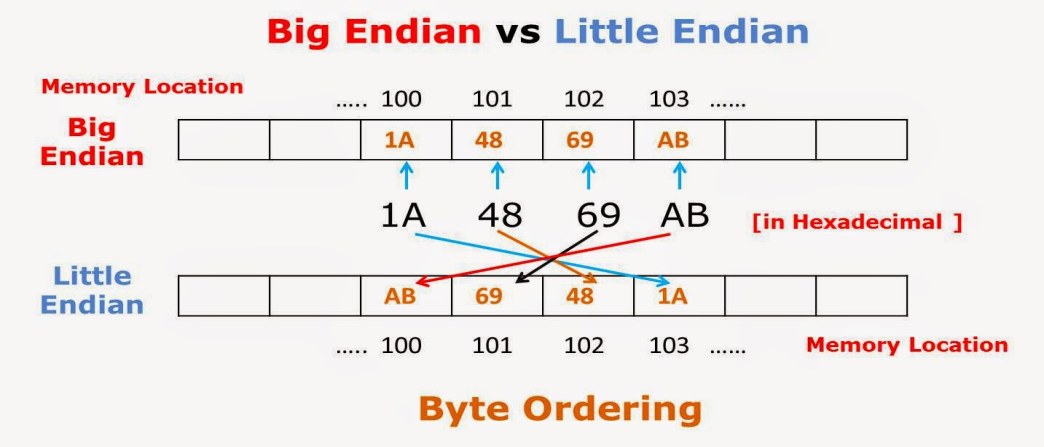
This helps reduce compilation time a lot

<https://www.youtube.com/watch?list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb&v=eSI4wctZUto>

# Byte Ordering – Big Endian vs Little Endian

## What Are Big Endian and Little Endian?

<https://www.youtube.com/watch?v=seZLUbgbB7Y>

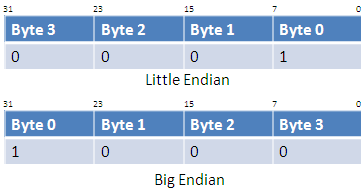


Little and big endian are two ways of storing multi-byte data-types (int, float, etc.).

* In little endian machines, the last byte of binary representation of the multi-byte data type is stored first.
* In big endian machines, the first byte of binary representation of the multi-byte data type is stored first.

In a multi-byte data type, the right most byte is called least significant byte (LSB), while the left most byte is called most significant byte (MSB).

How they differ depends on how they manage byte content in 32/64bit register. A little-endian CPU manage lower bytes in lower bit offsets which uniformly matches from LSB towards MSB. Whereas a big-endian processor manages bytes in reverse order. Lowest byte goes in most significant bits and gradually upper bytes are managed in lower bits. Below diagram shows how bytes are arranged in 32bit registers.



## How to See Memory Representation of Multi-Byte Data Types on Your Machine?

There is a number of ways for determining endianness of your machine. Here is one quick way of doing the same.

#include <stdio.h>

int main() {

unsigned int i = 1;

char\* c = (char\*)&i;

if (\*c == 1)

printf("Little endian");

else

printf("Big endian");

getchar();

return 0;

}

In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced, it will contain only first byte of integer. If machine is little endian then \*c will be 1 (because last byte is stored first) and if machine is big endian then \*c will be 0.

## Does Endianness Matter for Programmers?

Most of the times, compiler takes care of endianness (compiler and assembler take care of the bitwise and shifting operations). However, endianness becomes an issue in following cases.

### Network Programming

Suppose you write integers to file on a little-endian machine and you transfer this file to a big-endian machine. Unless there is little endian to big endian transformation, big endian machine will read the file in reverse order. You can find such a practical example here.

Standard byte order for networks is big endian, also known as network byte order. Before transferring data on network, data is first converted to network byte order (big endian).

### Type Casting

Sometimes it matters when you are using type casting, below program is an example.

#include <stdio.h>

**int** main() {

unsigned char arr[2] = {0x01, 0x00};

unsigned short int x = \*(unsigned short int\*) arr;

printf("%d", x);

getchar();

return 0;

}

In the above program, a char array is type-casted to an unsigned short integer type. When I run above program on little endian machine, I get 1 as output, but if I run it on a big-endian machine, I get 256. To make programs endianness independent, above programming style should be avoided.

## What Are Bi-Endians?

Bi-endian processors can run in both modes little and big endian.

Intel based processors are little endians. ARM processors were little endians. Current generation ARM processors are bi-endian.

Motorola 68K processors are big endians. PowerPC (by Motorola) and SPARK (by Sun) processors were big endian. Current version of these processors are bi-endians.

## Does Endianness Affects File Formats?

File formats which have 1 byte as a basic unit are independent of endianness (e.g., ASCII files). Other file formats use some fixed endianness format (e.g., JPEG files) are stored in big endian format.

# Multi-Threading and Multi-Processing

Refer to tutorial *Linux/Process and Thread.docx*

# Network

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjH_zM5LnovnoaKlXML5qh17>

# Algorithm

## String

### Split string using delimiter

<https://www.techiedelight.com/split-string-cpp-using-delimiter/>

# Tips

## Console

### Display Unicode characters on Windows CMD

* Still use default locale for Unicode characters which is English.
* Input is still in UTF-8.
* No need to use wide string

Before printing to console, set CMD page code to 65001 by adding the following code:

#include <windows.h>

SetConsoleOutputCP(65001);

More details: <https://stackoverflow.com/a/17177904>

### Aligned columns in table with std::cout

<https://stackoverflow.com/a/15674259>

# Error Codes

## Invalid use of incomplete type, or Incomplete type is not allowed

Root cause: Use a type that has only been forward declared, but not fully defined.

More details, check [this session](#_Forward_Declaration).

## Cannot call member function without object

Root cause: Call a member function of a class without an instance of that class.