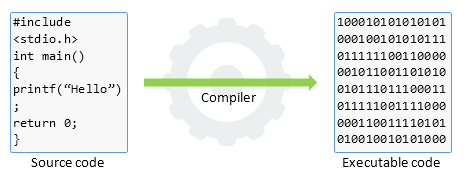
# The C/C++ Compilation Process

## What Is Compilation?

The process of translating source code written in high-level language to low-level machine code is called *compilation*. The compilation is done by a special software known as [compiler](https://codeforwin.org/2017/05/compiler-and-its-need.html). The compiler checks source code for any syntactical or structural error, and generates object code with extension .obj (in Windows) or .o (in Linux) if source code is error-free.

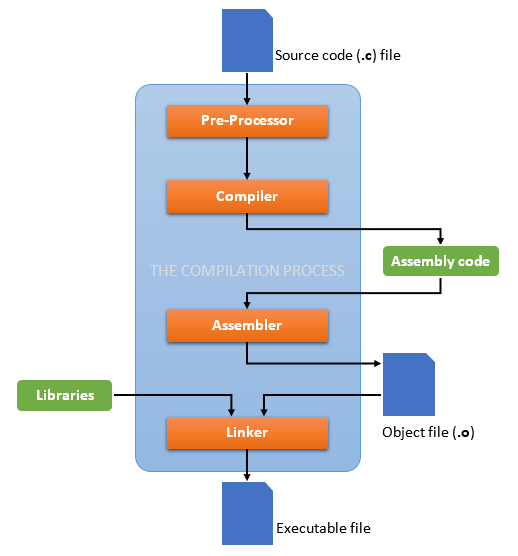


## The C/C++ Compilation

The entire C/C++ compilation process is broken to four stages.

* Pre-processing
* Compilation
* Assembling
* Linking

The below image describes the entire C/C++ compilation process:



To take a deep dive inside the C/C++ compilation process, let’s compile a C program. Create a text file named helloworld.c without following content:

#include <stdio.h>

int main()

{

printf("Hello, World!");

return 0;

}

To compile the above program, open command prompt and use below command:

gcc -save-temps helloworld.c -o helloworld

The -save-temps option will preserve and save all temporary files created during the C/C++ compilation. It will generate four files in the same directory namely:

// If on Linux:

helloworld.i (generated by pre-processor)

helloworld.s (generated by compiler)

helloworld.o (generated by assembler)

helloworld (generated by linker)

// If on Windows:

helloworld.i (generated by pre-processor)

helloworld.s (generated by compiler)

helloworld.obj (generated by assembler)

helloworld.exe (generated by linker)

Now let's look into these files and learn about different stages of compilation.

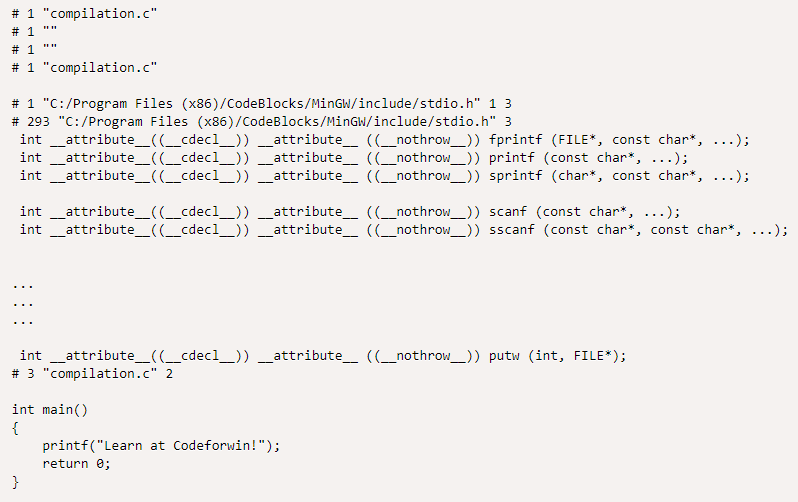
### Pre-Processing

Pre-processor is a small software that performs below tasks:

* Remove comments from the source code.
* Expansion of included header files.
* Macro expansion.

After pre-processing, a temporary with .i extension is generated. Since, it inserts contents of header files to the source code file, this generated file has a larger size than the original source code file.

Here is an extract of compilation.i file:



You can notice that the statement #include <stdio.h> is replaced by its contents. Comment before the #include line is also trimmed.

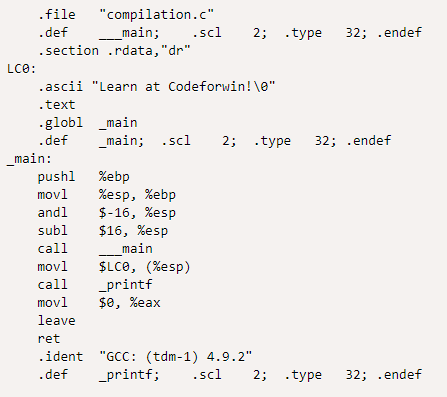
### Compilation

In next phase, the compiler performs following tasks:

* Check the program for syntax errors.
* Translate the file into assembly language (intermediate code).
* Optionally optimize the translated code for better performance.

After compiling, an ***intermediate code file*** (in assembly language) with .s extension is generated.

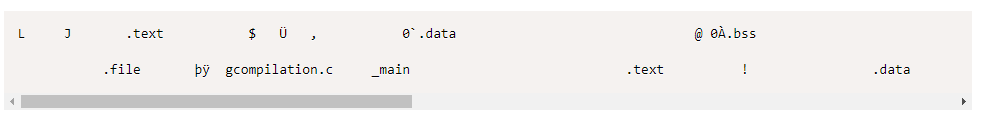
Let us look into compilation.s file:



### Assembling

Assembler accepts the intermediate code file and translates to machine code. After successful assembling, it generates .o file (on Linux) or .obj file (on Windows) known as ***object file***.

In our case, a compilation.o file is generated. It is encoded in machine language and cannot be viewed using text editors. However, if you still open it in a text editor, it looks like:



### Linking

Finally, the linker comes in action and performs the final task of compilation process. It accepts the object file. It links all the function calls with their original definition; that means the function printf() gets linked to its original definition.

The linker can generate one of following files based on your configuration:

* ***Executable file*** (no extension on Linux, or .exe on Windows).
* ***Static library*** (.a on Linux – also called *archive library*, or .lib on Windows).
* ***Dynamic library*** (.so on Linux – also called *shared object library*, or .dll on Windows).

**Static Linking and Dynamic Linking**

|  |  |
| --- | --- |
| **Static linking** | **Dynamic linking** |
| Done by the linker in the final step of the compilation. | Done at run time by the OS. |
| Statically linked files consume more disk and memory as all the modules are already linked. | Only one copy of the reference module is stored which is used by many programs, thereby saving memory and disk space. |
| All the library modules are copied to the final executable image. When the program is loaded, the OS keeps only a single file in the memory which contains both the source code and the referencing libraries. | Only the names of external or shared libraries are kept in the memory. Dynamic linking lets many programs use single copy of executable module. |
| If external source program is changed then they have to be re-compiled and re-linked. | Only a single module needs to be updated and re-compiled. This is one of the greatest advantages dynamic linking offers. |
| Statically linked programs are faster than their dynamic counterpart. | Dynamically linked programs are slower than their static counterpart. |
| Since the statically linked file contains every package and module, no compatibility issues occur. | Since the library files are separately stored there may be compatibility issues (say one library file is compiled by new version of compiler). |
| Statically linked programs always take constant load time. | The time is variable in dynamically linked programs. |

# Data Type

While writing program in any language, you need to **use variables to store information**. Variables are nothing but reserved memory locations to store values. This means that when you create a variable, you reserve some space in memory.

## Built-In Data Types

C++ offers a rich assortment of both **built-in** and **user-defined** data types. Following table lists down seven basic primitive built-in C++ data types:

|  |  |
| --- | --- |
| **Type** | **Keyword** |
| Boolean | bool |
| Character | char |
| Integer | int |
| Floating point | float |
| Double floating point | double |
| Valueless | void |
| Wide character | wchar\_t |

Several of these basic types can be modified using one or more of these type modifiers:

* signed
* unsigned
* short
* long

The following table shows how much memory a variable type takes to store the value in memory, as well as the maximum and minimum value which can be stored in such type of variables.

|  |  |  |
| --- | --- | --- |
| **Type** | **Typical Bit Width** | **Typical Range** |
| *char* | 1 byte | -127 to 127 or 0 to 255 |
| *unsigned char* | 1 byte | 0 to 255 |
| *signed char* | 1 byte | -127 to 127 |
| *int* | 4 bytes | -2147483648 to 2147483647 |
| *unsigned int* | 4 bytes | 0 to 4294967295 |
| *signed int* | 4 bytes | -2147483648 to 2147483647 |
| *short int* | 2 bytes | -32768 to 32767 |
| *unsigned short int* | Range | 0 to 65,535 |
| *signed short int* | Range | -32768 to 32767 |
| *long int* | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| *signed long int* | 4 bytes | same as long int |
| *unsigned long int* | 4 bytes | 0 to 4,294,967,295 |
| *float* | 4 bytes | +/- 3.4e +/- 38 (~7 digits) |
| *double* | 8 bytes | +/- 1.7e +/- 308 (~15 digits) |
| *long double* | 8 bytes | +/- 1.7e +/- 308 (~15 digits) |
| *wchar\_t* | 2 or 4 bytes | 1 wide character |

**Note**: The size of variables might be different from those shown in the above table, depending on the **register length in the processor of the computer** you are using.

To make integer fixed in size across all machines, C++99 introduces [Standard Integer](https://www.youtube.com/watch?v=iX1uGr6Si0E&list=PL9IEJIKnBJjFJll5-WYYvTYspFtGUTB-_&index=4) (*uint8\_t*, *int64\_t*).

Following is the example, which will produce correct size of various data types on your computer.

[Live Demo](http://tpcg.io/iKNn78)

#include <iostream>

using namespace std;

int main()

{

cout << "Size of char: " << sizeof(char) << endl;

cout << "Size of int: " << sizeof(int) << endl;

cout << "Size of short int: " << sizeof(short int) << endl;

cout << "Size of long int: " << sizeof(long int) << endl;

cout << "Size of float: " << sizeof(float) << endl;

cout << "Size of double: " << sizeof(double) << endl;

cout << "Size of wchar\_t: " << sizeof(wchar\_t) << endl;

return 0;

}

This program uses pre-built function sizeof() to get size of various data types.

The above code produces the following result:

Size of char: 1

Size of int: 4

Size of short int: 2

Size of long int: 4

Size of float: 4

Size of double: 8

Size of wchar\_t: 4

## User-Defined Data Types

You can **create a new name for an existing type using**typedef as follows:

typedef type newname;

Example 1: The following tells the compiler that feet is another name for int:

typedef int feet;

Now you can create an integer variable called *distance*:

feet distance;

Example 2:

typedef long int \*pint32;

pint32 x, y, z;

x, y and z are all pointers to long int.

# Variables

## Global vs Static

### Global

#include <stdio.h>

int g\_var = 0;

int main()

{

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

return 0;

}

g\_var has global scope and is **visible everywhere in the file**. But to make it visible in the whole program (other modules and other files), you'd need an extern int g\_var.

A global variable lives until the end of file. And it is initialized upon program start to 0 without an explicit initialization.

### Static

Static variable can be declared outside of all functions or within a function. **If a static variable is declared outside of all functions, it will have global scope**. By contrast, **if it is declared within a function, it will have local scope within that function, but like global variables – it will live until the end of program**.

Static variables are initialized to 0 by the C standard if lacking an explicit initialization (although explicitly initializing them is a good practice).

Static variables are not accessible by other files using extern keywords.

#include <iostream>

void setNum()

{

static int num = 0;

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

num = 3;

}

int main()

{

setNum();

setNum();

return 0;

}

Output:

0 3

*Why “3”?*

Static and global variable differ a lot in their behavior to life and scope:

* Life of an object determines whether the object is still in the memory (of the process)
* Scope of an object is whether the variable can be called by its name at this position.

It is possible that an object is live, but not visible (not in scope). A static variable is such a kind of object. In particular, it is local in scope to its module in which it’s defined, but live throughout the program. Say for a static variable inside a function cannot be called from outside the function (because it's not in scope) but is alive and exists in memory. The next time this function is entered (within the same program) **the same chunk of memory would be accessed now retaining the variables old value and** **no new memory is allocated for this variable**.

# Functions

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**, 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:

[Live Demo](http://tpcg.io/XuXfLz)

#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. Consider the following example:

[Live Demo](http://tpcg.io/KoJLUN)

#include <iostream>

using namespace std;

int sum(int a, int b = 20) {

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

# 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

To declare an array, you must specify the type of the elements and the number of elements required by that array as follows:

type arrayName [arraySize];

In which, the arraySize must be an integer constant greater than zero andtype can be any valid C++ data type.

This is called a single-dimension array. For example, to declare a 10-element array called balance of type double, use this statement:

double balance[10];

## 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};

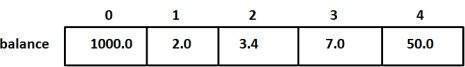
Or

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

Or initialize all elements as 0.0

double balance[] = {0.0};

The pictorial representation of this array is:



## Accessing Array Elements

An element is accessed by **indexing the array name**. This is done by placing the index of the element within square 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:

[Live Demo](http://tpcg.io/QRe4fY)

#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;

// output each array element's value

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

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

}

return 0;

}

This program makes use of setw function to format the output. When the above code is compiled and executed, it produces the following result:

Element Value

0 100

1 101

2 102

3 103

4 104

5 105

6 106

7 107

8 108

9 109

## Important Array Concepts

|  |  |
| --- | --- |
| **Concept** | **Description** |
| Multi-dimensional arrays | C++ supports multidimensional arrays. The simplest form of the multidimensional array 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 general form of a multidimensional array declaration:

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

For example, the following declaration creates a three-dimensional integer array:

int threeDim[5][10][4];

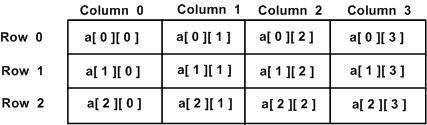
#### Two-Dimensional Arrays

The simplest form of multidimensional arrays is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional 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 two-dimensional array 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 Two-Dimensional Arrays

Multi-dimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and 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 Two-Dimensional Array Elements

An element in 2-dimensional 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.

[Live Demo](http://tpcg.io/IgS5rD)

#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;

}

When the above code is compiled and executed, it produces the following result:

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 an 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 legitimate 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.

Below is the example to show all the concepts discussed above:

[Live Demo](http://tpcg.io/KlljEG)

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

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

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

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

}

return 0;

}

When the above code is compiled and executed, it produces the following result:

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 an example to demonstrate why it’s wrong to pass an entire array as an argument to a function:

void func(int inArr[]) {

size\_t size = sizeof(inArr);

printf("%d\n", size);

}

void main() {

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

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

printf("%d\n", size);

func(arr);

}

Output:

12

4

In C++ functions, arrays are [**passed by pointer**](#_2s8eyo1) by default. For how arrays are passed by reference, check [here](https://www.quora.com/In-C++-functions-why-are-arrays-passed-by-reference-by-default/answer/M-Santosh-Dora-1).

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways. These methods produce similar results because each tells the compiler that an integer pointer is going to be received.

#### Way 1: Parameters as a pointer

void myFunction(int \*param) {

...

}

#### Way 2: Parameters as a sized array

void myFunction(int param[10]) {

...

}

#### Way 3: Parameters as an un-sized array

void myFunction(int param[]) {

...

}

Now, consider the following function, which will take an array as an argument along with another argument:

#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 as an argument to a function. However, you can return a pointer to the first element of the array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example:

int\* myFunction() {

...

}

Another point to remember is that C++ does NOT advocate to return the address of a local array to outside of the function, so you would have to define the local array as *static*.

Now consider the following function, which will generate 10 random numbers and return them using an array and call this function as follows:

[Live Demo](http://tpcg.io/CIFH0A)

#include <iostream>

#include <ctime>

using namespace std;

int\* getRandom()

{

static int r[10];

// set the seed

srand((unsigned)time( NULL ));

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

r[i] = rand();

cout << r[i] << endl;

}

return r;

}

int main()

{

int \*p;

p = getRandom();

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

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

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

}

return 0;

}

When the above code is compiled together and executed, it produces result something as follows:

624723190

1468735695

807113585

976495677

613357504

1377296355

1530315259

1778906708

1820354158

667126415

\*(p + 0): 624723190

\*(p + 1): 1468735695

\*(p + 2): 807113585

\*(p + 3): 976495677

\*(p + 4): 613357504

\*(p + 5): 1377296355

\*(p + 6): 1530315259

\*(p + 7): 1778906708

\*(p + 8): 1820354158

\*(p + 9): 667126415

# 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.

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

[Live Demo](http://tpcg.io/fQ1InE)

#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;

Following are the valid pointer declaration:

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** 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.

[Live Demo](http://tpcg.io/gTdFv7)

#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. |
| Pointer Arithmetic | There are four 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 |  |

### Null Pointer

It is always a **good practice to assign NULL to a pointer** when declaring it if you do not have exact address to be assigned. A pointer that is assigned NULL is called a null pointer. 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:

[Live Demo](http://tpcg.io/0sA3lU)

#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 of the operating systems, 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 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 variant.

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.

### Pointer Arithmetic

As you understood, 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 four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, 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:

[Live Demo](http://tpcg.io/g0pNAF)

#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:

[Live Demo](http://tpcg.io/ujgz9K)

#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].

[Live Demo](http://tpcg.io/T8eHu9)

#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. Consider the following program:

[Live Demo](http://tpcg.io/Z3TAMf)

#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, consider the following program:

#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:

[Live Demo](http://tpcg.io/JdrZow)

#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 three integers which will be stored in an array of pointers:

[Live Demo](http://tpcg.io/qgpsAZ)

#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:

[Live Demo](http://tpcg.io/0kooHI)

#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 as such. This is done by placing an additional asterisk 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 asterisk operator be applied twice, as is shown below in the example:

[Live Demo](http://tpcg.io/KEQl2G)

#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:

[Live Demo](http://tpcg.io/6sMZid)

#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:

[Live Demo](http://tpcg.io/XgyLBa)

#include <iostream>

using namespace std;

// function declaration:

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

As we have seen in last chapter about how C++ allows to return an array from a function, similar way C++ allows you 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. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

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

Let us try the following example to understand the concept of this pointer:

[Live Demo](http://tpcg.io/y4Ar2U)

#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;

breadth = b;

height = h;

}

double Volume() {

return length \* breadth \* height;

}

int compare(Box box) {

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

}

private:

double length; // Length of a box

double breadth; // Breadth 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

# 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 Variable vs Pointer Variable

Reference variables are often confused with pointer variable, but three 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 legitimate 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.

## Creating References in C++

For example:

int i = 17;

We can declare reference variables for i as follows.

int& r = i;

Consider the example below:

[Live Demo](http://tpcg.io/SS8zU0)

#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;

return 0;

}

Output:

Value of i: 5

Value of i reference: 5

## Important Concepts

### References as 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:

[Live Demo](http://tpcg.io/mbXI33)

#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

### Reference as Return Value

A C++ program can be made easier to read and maintain by using references rather than pointers. 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:

[Live Demo](http://tpcg.io/xI2Oqo)

#include <iostream>

#include <ctime>

using namespace std;

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

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

When returning a reference, be careful that the object being referred to does not go out of scope. So, it is not legal 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

}

# 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 **one-dimensional arrays of characters,** which are 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**.

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 secure 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**

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

In the following program, "GfG" is stored in a shared read only location, but pointer str is stored in a read-write memory. **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";

**Note 2: const char\* ptr vs char\* const ptr**

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";  // or char\* src = "abc";  char dest[10]; | dest = src;  // or dest = (char)src; | memcpy(dest, src, strlen(src));  // or memcpy(dest, src, 10); |
| char src[] = "abc";  char\* dest;  // or const char\* dest; |  | dest = src; |

**Pass char\* by references:**

The series of examples below yield one of following outputs:

SUCCESS: a is 2 and b is xyz

FAILURE: a is 2 and b is abc

#include <iostream>

using namespace std;

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

{

\*outA = 2;

// outB = (char\*)malloc(sizeof(char)\*10);

// if add this line, output of outB is still "abc"

outB = "xyz";

}

int main()

{

int a = 1;

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

// const char\* b = (char\*)calloc(10, sizeof(char)); b = "abc";

// if replace above line by this, output of outB is still "abc"

setAB(&a, b);

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

}

Output:

a is 2 and b is abc

#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

#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

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

// if replace above line this, 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

#include <iostream>

using namespace std;

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

{

\*outA = 2;

// outB = (char\*)"xyz";

// if using this, output of outB is still "abc"

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

}

int main()

{

int a = 1;

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

// if put this on setAB(), output of outB is still "abc"

b = (char\*)"abc"; // in C, can use b = "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 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 and map). But generally, we use a simple for loop for iterating over the characters and index them using []operator.

**Library: <string>**

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

**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:

[Live Demo](http://tpcg.io/upuWFC)

#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;

}

When the above code is compiled and executed, it produces result something as follows:

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. |
| 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). |
| 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). |
| 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:

Instead of replacing:

**char** cResponse;

Same as:

**CHAR** cResponse;

with

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

Same as:

**WCHAR** cResponse;

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

#include<TCHAR.H>

**TCHAR** cResponse;

This way, when your project is being compiled as Unicode, 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.

That’s because:

#ifdef \_UNICODE

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

#else

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

#endif

The macro \_UNICODE is defined; when you set Character Set to "*Use Unicode Character Set*", TCHAR would mean wchar\_t. When Character Set if set to "*Use Multi-Byte Character Set*", TCHAR would mean char.

### 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.

# 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): Classes can override the ++ operator, and because of dynamic linking and potential side effects, it becomes impossible to know when to optimize away the copy. So, **use pre-increment whenever you can**.

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)

differs from:

vector<int> v(n);

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

# File Streams

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.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

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**.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

void fstream::close();

## Reading from a File

You read information from a file into your program using 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.

## Writing to a File

While doing C++ programming, you write information to a file from your program using 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.

## 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 onto the screen:

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

int main()

{

string data;

// open a file in write mode

ofstream writeFile;

writeFile.open("input.txt");

// 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();

// open a file in read mode

ifstream readFile;

readFile.open("input.txt");

// read the file

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

// because we want to read white space, tab and eof in the file,

// we 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;

}

When the above code is compiled and executed, it produces the following sample input and 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.** Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

* Title
* Author
* Subject
* Book ID

## 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];

The *structure tag* is optional and each member definition is a normal variable definition, such as int i, float f or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables, but it is optional.

Here is the way you would declare 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 \_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 keyword 'typedef':**

#include <iostream>

using namespace std;

struct \_Student {

string name;

int id;

};

int main()

{

struct \_Student student; // Note: Still work if do not have "struct"

student.name = "abc";

student.id = 1;

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

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

}

**With keyword 'typedef':**

#include <iostream>

using namespace std;

typedef struct \_Student {

string name;

int id;

} Student;

int main()

{

Student student; // Note: Syntax error if using "struct Student student;"

student.name = "abc";

student.id = 1;

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

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

#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;

}

## 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

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.

Another way to declare this struct:

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.

# 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 Vector

Vector is a **sequence container** (a container holds data of same type) and a perfect replacement for the old C-style arrays. They are same as dynamic arrays with the ability to **resize themselves automatically** when an element is inserted or deleted.

Vector elements are placed in contiguous storage so that they can be accessed and traversed using [**iterators**](#_35nkun2). We can also access any element of a vector using [] operator, exactly like an array.

In vectors, **data is inserted at the end** using push\_back()**or in the middle** using insert(). Inserting at the end takes differential time (as sometimes there may be a need of extending the array), but inserting in the middle is linear in time. Removing the last element using pop\_back() takes only constant time because no resizing happens.

**Library:**

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

## Vectors vs Dynamic Arrays

Let's start small and take the example of an array of integers. If you used plain arrays, you had either a static or a **dynamic array**:

size\_t size = 10;

int sArray[10];

int \*dArray = new int[size];

// do something with them:

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

sArray[i] = i;

dArray[i] = i;

}

// don't forget to delete dArray when you're done

delete[] dArray;

Let's do the same thing using a vector:

#include <vector>

//...

size\_t size = 10;

std::vector<int> v(size); // make room for 10 integers,

// and initialize them to 0

// do something with them:

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

v[i] = i;

}

// no need to delete anything

As you see, vectors combine the advantages of both static and dynamic arrays because they **take a non-const size parameter** (like dynamic arrays) and **automatically delete the used memory** (like static arrays).

**Iterator Library:**

<https://www.geeksforgeeks.org/iterators-c-stl/>

## Different Ways to Iterate Through a Vector

vector<int> v;

// Way 1: index based

for(size\_t i = 0; i < v.size(); ++i) {

cout << v[i] << endl;

}

// Way 2: iterator

vector<int>::iterator it;

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

cout << \*it << endl;

}

// Way3: range-based for loop

for(auto&& x: v) {

cout << x << endl;

}

More ways: <https://stackoverflow.com/a/2395307>

**Detailed explanation on how to use vectors:**

<https://www.codeguru.com/cpp/cpp/cpp_mfc/stl/article.php/c4027/C-Tutorial-A-Beginners-Guide-to-stdvector-Part-1.htm>

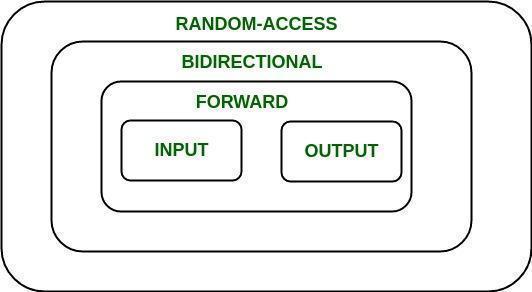
# STL Iterator

## Definition

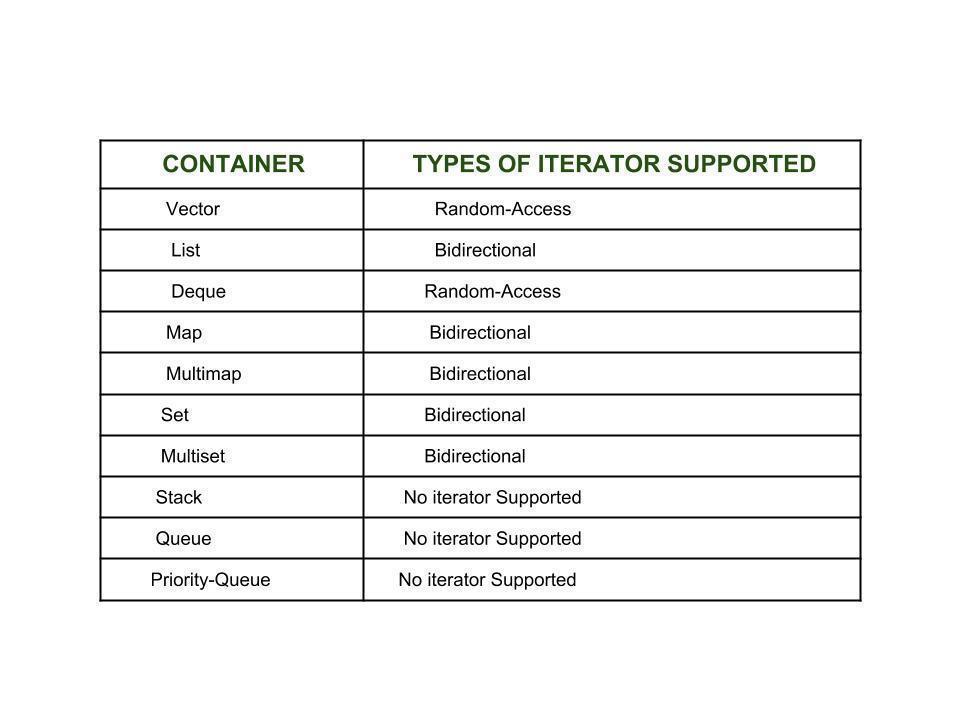
An iterator is an object (similar to a pointer) that **points to an element inside the container**. We can use iterators to move through the contents of the container to access a particular content at a particular location.

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 each one of these iterators is not supported by all the containers in STL. Different containers support different iterators as given below:



## 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] |

*Where X is an iterator type, a and b are objects of this iterator type, t is an object of the type pointed by the iterator type, and n is an integer value.*

## 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 **every STL container provides an iterator**.

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 i;

// Inserting element using iterators

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

if (i == v.begin()) {

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

}

} // v = {5 1 2 3}

// Deleting element using iterators

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

if (i == v.begin() + 1) {

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

}

} // v = {5 2 3}

// Accessing the elements using iterators

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

cout << \*i << " ";

}

return 0;

}

Output:

5 2 3

# Function Overloading vs Function Overriding

## Function Overloading

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**.

#include <iostream>

using namespace std;

// overloaded functions

void test(int);

void test(float);

void test(int, float);

int main() {

  int a = 5;

  float b = 5.5;

  // Overloaded functions

  test(a);

  test(b);

  test(a, b);

  return 0;

}

// Method 1

void test(int var) {

  cout << "Integer number: " << var << endl;

}

// Method 2

void test(float var) {

  cout << "Float number: "<< var << endl;

}

// Method 3

void test(int var1, float var2) {

  cout << "Integer number: " << var1;

  cout << " and float number:" << var2;

}

Output:

Integer number: 5

Float number: 5.5

Integer number: 5 and float number: 5.5

## 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**.

#include<iostream>

using namespace std;

class BaseClass

{

public:

virtual void Display() {

cout << "\nThis is Display() method of BaseClass";

}

void Show() {

cout << "\nThis is Show() method of BaseClass";

}

};

class DerivedClass : public BaseClass

{

public:

// Overriding method - new working of base class's display method

void Display() {

cout << "\nThis is Display() method of DerivedClass";

}

};

int main()

{

DerivedClass dr;

// Without this line, virtual void Display() of BaseClass will be executed

// instead of void Display() of DerivedClass

BaseClass &bs = dr;

bs.Display();

dr.Show();

}

Output:

This is Display() method of DerivedClass

This is Show() method of BaseClass

# Classes and Objects

The main purpose of C++ is to add object orientation to C, and **classes are the central feature of C++**. They’re also called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called **members** of the class.

## Class Definition

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword class followed by the class name and the class body (enclosed by a pair of curly braces and followed by a semicolon). For example, we defined the Box data type using the keyword class as follows:

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

## Class Instance (Object)

We declare instances of a class (or *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 Data Members

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

[Live Demo](http://tpcg.io/JeEWd6)

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth 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.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

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

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

// volume of box 2

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

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 is a function that has its definition or its prototype within the class definition like any other variable. |
| [**Class Access Modifiers**](https://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm) | 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 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**](https://www.tutorialspoint.com/cplusplus/cpp_copy_constructor.htm) | The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. |
| [**Friend Functions**](https://www.tutorialspoint.com/cplusplus/cpp_friend_functions.htm) | A friend function is permitted full access to private and protected members of a class. |
| [**Inline Functions**](https://www.tutorialspoint.com/cplusplus/cpp_inline_functions.htm) | With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function. |
| [**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**](https://www.tutorialspoint.com/cplusplus/cpp_static_members.htm) | 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. |

### Static Members of Classes

#### Static Variables

Variables declared as static are shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. There **cannot be multiple copies** of the same static variable for different objects. So we **cannot put it in the class definition, but it can only be initialized outside the class** as done using the scope resolution operator (::). Also because of this reason, static variables **cannot be initialized in constructors**.

The below program FAILS TO COMPILE because we cannot create multiple copies of the static variable for multiple objects:

#include<iostream>

using namespace std;

class A

{

public:

static int i;

A()

{

// Do nothing

};

};

int main()

{

A obj1;

A obj2;

obj1.i = 2;

obj2.i = 3;

cout << obj1.i << " " << obj2.i;

}

But this program WORKS WELL:

#include<iostream>

using namespace std;

class A

{

public:

static int i;

A()

{

// Do nothing

};

};

// i is initialized explicitly by the user

int A::i = 1;

int main()

{

A obj;

cout << obj.i;

}

#### Static Member Functions

Just like static variables of classes, static member functions don’t depend on any object of class (can be called even if no objects of the class exist). We are allowed to invoke a static member function using the object and the ‘.’ operator, but it is recommended to do that using the ‘::’ operator.

Note: 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 or member functions of the class, and they cannot access to the this pointer of the class, too.

#include<iostream>

using namespace std;

class A

{

public:

// static member function

static void printMsg()

{

cout << "Welcome to A!";

}

};

// main function

int main()

{

// invoking a static member function

A::printMsg();

}

### Constant Member Functions

A function becomes constant when 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.

Following is a simple example of constant function.

...

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;}

};

**Note**: When a function is declared as const, it 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.

...

class Test {

int value;

public:

...

int getValue() {return value;}

};

int main() {

const Test t;

// COMPILE ERROR: passing 'const Test' as 'this' argument of 'int

// Test::getValue()' discards qualifiers

cout << t.getValue();

return 0;

}

# 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 completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base class**, and the new class is referred to as the **derived** **class**.

The idea of inheritance implements the IS-A relationship. For example, mammal IS-A animal, dog IS-A mammal, hence dog IS-A animal as well.

### 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, and *base-class* is the name of a previously defined class. If the *access-specifier* is not used, then it is private by default.

Consider a base class Shape and its derived class Rectangle as follows:

#include <iostream>

// Base class

class Shape {

    public:

        void setWidth(int w) {

            width = w;

        }

        void setHeight(int h) {

            height = h;

        }

    protected:

        int width;

        int height;

};

// Derived class

class Rectangle : public Shape {

    public:

        int getArea() {

            return (width \* height);

        }

};

int main(void) {

    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 Control

**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 |

A derived class inherits all base class methods with the following exceptions:

* Constructors, destructors and copy constructors of the base class.
* Overloaded operators of the base class.
* The friend functions of the base class.

### Type of Inheritance

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.

### Multiple Inheritance

**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(void) {

    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

## Polymorphism

The word *polymorphism* means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they 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 what a virtual function is and how polymorphism works, let consider the following 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 incorrect 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 you can see, 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 (in this example, no parameter), but with different implementations. ﻿This is called [overriding](#_1ksv4uv).

From the example above, you can define polymorphism as "**Polymorphism means writing general code to work with different objects without knowing their exact types**."

### 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 do 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 is possible to include a virtual function in a base class with no meaningful definition. To do that, change the virtual function in the base class to the following:

Example 1: virtual void area() = 0;

Example 2: virtual void fun(int x) = 0;

The "= 0" tells the compiler that the function has no body and above virtual function is called pure virtual function. ﻿

## 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).

For example, a database system hides certain details of how data is created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

<https://www.tutorialspoint.com/cplusplus/cpp_data_abstraction.htm>

## Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables, but object-oriented programming provides you framework to place the data and relevant functions together in the same object.

<https://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm>

# Preprocessors

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. Although each one is available for use in programming, the predefined macros should not be directly modified.

|  |  |
| --- | --- |
| **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\_\_ | The current function name (including class name) as a string literal. |
| \_\_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:

[Live Demo](http://tpcg.io/BMJM0C)

#include <stdio.h>

int main() {

printf("File: %s\n", \_\_FILE\_\_ );

printf("Date: %s\n", \_\_DATE\_\_ );

printf("Time: %s\n", \_\_TIME\_\_ );

printf("Line: %d\n", \_\_LINE\_\_ );

printf("ANSI: %d\n", \_\_STDC\_\_ );

}

Output:

File: test.c

Date: Jun 2 2012

Time: 03:36:24

Line: 8

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:

[Live Demo](http://tpcg.io/gXEa63)

#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()).

#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()

**Note:** Below program will not work with GCC compilers. This happens because GCC does not supports #pragma startup or exit. However, you can use the below code for a similar output on GCC compilers.

#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;

}

### #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:

[Live Demo](http://tpcg.io/qLYOKm)

#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:

[Live Demo](http://tpcg.io/2ZlJsc)

#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

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 is one of the sections of a program in an object file or in memory, which **contains executable instructions**.

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, the C compiler, the 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 is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

This segment can be further classified into initialized **read-onl**y area and initialized **read-write** area.

For instance the global string defined by char s[] = "hello world" or static int debug = 1 outside the main (i.e. global) is stored in initialized read-write area. But a 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." Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing.

For example, a global variable declared int i or static int j would be contained in the BSS segment.

## Stack

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.

The biggest advantage of using stacks to store variables is that memory is managed for you. You don't have to allocate memory by hand, or free it once you don't need it any more. What's more, because the CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

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). A common bug in C programming is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function (i.e., after that function has exited).

However, the biggest disadvantage of using stacks is that there is a limit (varies with OS) on the size of variables that can be stored on the stack. Luckily, this is not the case for variables allocated on the heap.

## Heap

The heap is a region of the computer's memory. It is not managed automatically for you and not as tightly managed by the CPU. But it is a more free-floating region of memory (and is larger).

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.

Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use *pointers* to access memory on the heap.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. So, heap variables are essentially global in scope.

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 to use heap?

If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack.

However, 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), then you should allocate it on the heap. In addition, 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), then you will likely need to allocate them on the heap.

# Dynamic Memory Management

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts: [stack and heap](#_1y810tw).

## 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);

}

**You might not know**: new vs malloc()

In C++, you can use the [malloc()](#_2xcytpi) in C to do the same task, but it is not recommended. The main advantage of new over malloc() is that it doesn't just allocate memory, but it constructs objects which is prime purpose of C++.

Another difference between new and malloc() is that new returns data with a specific type. Wheares, malloc() lets it void.

### 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

Let us put above concepts to the following example to show how new and delete work:

[Live Demo](http://tpcg.io/YFq73r)

#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

**Good practive**: Always set a pointer to null after deleting the object it points to. See [why](https://stackoverflow.com/questions/1931126/is-it-good-practice-to-null-a-pointer-after-deleting-it).

### Notes

#### For Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax, you can allocate memory dynamically as shown below:

char\* pvalue = new char[20]; // Request memory for the variable

To remove the array that we have just created the statement would look like this:

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).

#### For Objects

Consider the following example:

[Live Demo](http://tpcg.io/qxQQ36)

#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!

## 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

# 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**.

### What Cause Segmentation Faults?

**1.** Calling memset() as shown below would cause a program to segfault:

memset((char\*)0x0, 1, 100);

**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;

}

In this case, 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 result in a segfault if that memory location lies outside the memory segment where foo resides. Even if it doesn't cause a segfault, it is still a bug.

Case B

// Illegal memory access because no memory is allocated for foo2

float \*foo, \*foo2;

foo = (float\*)malloc(1000);

foo2[0] = 1.0;

In this case, allocation of memory for variable foo2 has been overlooked, so foo2 will point to a random location in memory. Accessing foo2[0] will likely result in a segfault.

**3.** Another common programming error that leads to segfaults is oversight in the use of pointers.

int foo = 0;

scanf("%d", foo);

// Note missing & sign ; correct usage would have been &foo

In this case, 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.

### How to Solve Segfault?

#### 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](https://kb.iu.edu/d/aqsj#limit), some segfault problems 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 GNU's well-known debugger 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.

For more, see [step-by-step example for using GDB within Emacs to debug a C or C++ program](https://kb.iu.edu/d/aqsy).

## Buffer Overruns

A buffer overrun (also called **buffer overflow**) 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. In some cases, the memory actually holds executable information (*heap overrun*). Instead of running the original executable code, the application ends up running the cracker's code. In other cases, the cracker overwrites the stack frame for the application (*stack overrun*).

The act of copying this data, using operations (such as *strcat*, *strcpy*) can create unanticipated results, which allows for system corruption.

In the best of cases, your application will abort with a core dump, [segmentation fault](#_3whwml4), or access violation. In the worst of cases, an attacker can exploit the buffer overrun by introducing and executing other malicious code in your process.

### Types and Causes of Buffer Overruns

#### Stack Overruns

A stack overrun occurs when **a buffer, which has been declared on the stack, is written to with more data than it was allocated to hold**. The less apparent versions of this error occur when unverified user input data is copied directly to a static variable, causing potential stack corruption.

void PrintHello(char\* name)

{

   char buf[10];

   strcpy(buf, "hello ");

   strcat(buf, name);

   puts(buf);

}

In the example above, a buffer buf is allocated on the stack and a string name is formatted based on the parameter passed to the function. The function is simply to copy the string *hello* into the temporary buffer buf, then append the string pointed to by the parameter.

On the surface, this function looks fine. However, the stack-allocated buffer can only hold 9 characters (10 if you count 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 Overruns

Heap overruns, like static buffer overruns, can lead to memory and stack corruption. Because heap overruns occur **in heap** memory rather than on the stack, some people consider them to be less able to cause serious problems. However, heap overruns require real programming care and are just as able to allow system risks as static buffer overruns.

#### Array Indexing Errors

Careful bounds checking and index management will help prevent this type of memory overrun.

### Preventing Buffer Overruns

The example above is typical of the kind that lets buffer overruns occur. In essence, these are functions that fill a user-allocated buffer. If your code uses any of these functions, carefully check to ensure that you are allocating sufficiently large buffers.

For example, if your code uses strcat, a potential buffer overrun is waiting to happen, so you should replace this with strncat and provide the maximum number of characters that should be copied.

For various methods to prevent buffer overruns: <http://www.informit.com/articles/article.aspx?p=169527>

## How to Detect Memory Leaks?

<https://web.stanford.edu/class/archive/cs/cs107/cs107.1186/guide/valgrind.html>

# Data Structures

## Linked List

Great video about how to implement linked list in C:

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFiudyP6wSXmykrn67Ykqib>

### Linked List vs Array

Both arrays and linked list can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.





**Drawbacks of arrays:**

1. The size of the arrays is fixed: We must know the upper limit on the number of elements in advance. Also, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

2. Inserting a new element to an array is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, …..].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

3. Deletion is also expensive with arrays until unless some special techniques are used.

For example, to delete 1010 in id[], everything after 1010 has to be moved.

**Linked list provides following two advantages over arrays:**

1. Dynamic size

2. Ease of insertion/deletion

**But linked lists have following drawbacks:**

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So, we cannot do binary search with linked lists.

2. Extra memory space for a pointer is required with each element of the list.

3. Arrays have better cache locality that can make a pretty big difference in performance.

### Why double pointers are used in linked list?

It is clear that both methods (single pointer and double pointer) lead to the **same result**. The only difference is **what will be changed afterward**.

Double pointers are used as **arguments** of function when the function modifies and updates the linked list without needing to return the value (address or data) of the list again.

When using single pointers as arguments of function that modifiers and updates the linked list, we must return the value (address or data) of the list. Or else, the effect won’t be noticed.

Briefly, remember the simple C rule: If you want to **modify local variable of one function inside another function**, pass pointer to that variable. It is called "call by pointers". In this case, the pointer is C’s way of implementing "call by reference" when there is no reference variable.

For example, you want to add a new node before the head (first node) of the list, and hence, the pointer pointing to the first node will be then changed. When you exit this function, you want this change to reflect in the calling function and the following code in the main() function (suppose you call this function in the main()). In this case, you have to use a double pointer. One of them is to indicate that you are passing an address and another is to make the changes available to the calling function (to achieve call by reference).

## Stack

### Definition

Stack is a linear data structure that allows adding and removing elements in a specific order. In particular, every time an element is added, it goes on the top of the stack. The only element that can be removed is the one at the top of the stack. In other words, **the first item added to a stack will be the last item removed from it**. As a result, a stack is said to have "last in first out" behavior (or *LIFO*).

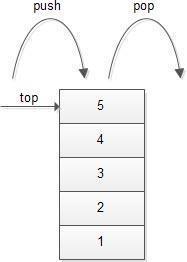
A typical example of using stack is function calling. A function calls another function, which in turn calls a third function; it's important that the third function return back to the second function rather than the first one.

*You might not know!*

The "callstack" is the term used for the list of functions either executing or waiting for other functions to return.

### Basic Operations

* **push**: Adds an item in a stack. If the stack is full, it is said to be an *Overflow* condition – Time complexity: O(1).
* **pop**: Removes an item from a stack. If the stack is empty, it is said to be an *Underflow* condition – Time complexity: O(1).
* **peek** or **top**: Returns a reference to the top most element of a stack – Time complexity: O(1).
* **size** – Returns the size of a stack – Time complexity: O(1).
* **isEmpty**: Returns true if stack is empty, else false – Time complexity: O(1).
* **isFull**: Returns true if stack is full, else false – Time complexity: O(1).



### Applications

* [Balancing of symbols](https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/)
* [Infix to Postfix](http://quiz.geeksforgeeks.org/stack-set-2-infix-to-postfix/) /Prefix conversion
* Redo-undo features at many places like editors, photoshop.
* Forward and backward feature in web browsers
* Used in many algorithms like [Tower of Hanoi,](https://www.geeksforgeeks.org/recursive-functions/)[tree traversals](https://www.geeksforgeeks.org/618/), [stock span problem](https://www.geeksforgeeks.org/the-stock-span-problem/), [histogram problem](https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/).
* Other applications can be Backtracking, [Knight tour problem](https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/), [rat in a maze](https://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/), [N queen problem](https://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/) and [sudoku solver](https://www.geeksforgeeks.org/backtracking-set-7-suduku/)
* In Graph Algorithms like [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/) and [Strongly Connected Components](https://www.geeksforgeeks.org/strongly-connected-components/)

### Implementation

There are two ways to implement a stack:

* Using array
* Using linked list

<https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>

### Stack in C++ STL

<https://www.geeksforgeeks.org/stack-in-cpp-stl/>

## Queue

### Definition

Queue is a linear data structure that allows adding and removing elements in a specific order To understand a queue, think of a cafeteria line: new people are added to the line at the back; the first person in line is served first, and the last person is served last. So, **in a queue the first item added to it will be the first item removed from it**. As a result, a queue is said to have "first in first out" behavior (or *FIFO*). That is opposite to a [stack](#_3as4poj).

*Note:*

Although the concept is simple, programming a queue is not as simple as programming a *stack*.

Let's go back to the example of the cafeteria line. Let's say one person leaves the line. Then what? Everyone in line must step forward, right? Now, imagine if only one person could move at a time. So, the second person steps forward to fill the space left by the first person, and then the third person steps forwards to fill the space left by the second person, and so on. Now imagine that no one can leave or be added to the line until everyone has stepped forward. You can see the line will move very slowly.

It is not difficult to program a queue that works, but it is **quite touch to make a queue that works fast**!

### Basic Operations

* **enqueue**: Adds an item to a queue. If the queue is full, it is said to be an *Overflow* condition – Time complexity: O(1).
* **dequeue**: Removes an item from a queue. If the queue is empty, it is said to be an *Underflow* condition – Time complexity: O(1).
* **front**: Get the front item from a queue – Time complexity: O(1).
* **rear** or **back**: Get the last item from a queue – Time complexity: O(1).



### Applications

Queue is used when things don’t have to be processed immediatly, but have to be processed in FIFO order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property makes queue useful in following scenarios.

* When a resource is shared among multiple consumers. For examples, CPU scheduling or disk scheduling.
* When data is transferred asynchronously (data isn’t necessarily received at same rate as sent) between two processes. For examples, IO buffers, pipes, file IO, etc.

### Implementation

1. Make an array and shift all the elements to accommodate enqueues and dequeues. This is slow, because with many elements, the shifting takes time.
2. Instead of shifting the elements, shifts the enqueue and dequeue points. Imagine that cafeteria line again. If the front of the line continually moves backwards as each person leaves the line, then people don't need to step forward or backwards, which saves time.

This method is much more complicated than the first one. Instead of keeping track of just the enqueue point (the "end"), we also need to keep track of the dequeue point (the "front"). This all gets even more complicated when we realize that after a bunch of enqueues and dequeues, the line will need to wrap around the end of the array. Think of the cafeteria line. As people enter and leave the line, the line moves farther and farther backwards, and eventually it will circle the entire cafeteria and end up at its original location.

<https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/>

<https://www.geeksforgeeks.org/queue-data-structure/>

### Queue in C++ STL

<https://www.geeksforgeeks.org/queuepush-and-queuepop-in-cpp-stl/>

### Different Types of Queues

#### Priority queue

#### Circular queue

## Associative Container

C++ has 4 different associative containers. With C++11, we get 4 additional ones:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Associate container | Sorted | Value | Several identical keys possible | Access time |
| [std::set](https://www.geeksforgeeks.org/set-in-cpp-stl/) | Yes | No | No | Logarithmic |
| std::unordered\_set (C++11) | No | No | No | Constant |
| [std::map](https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/) | Yes | Yes | No | Logarithmic |
| std::unordered\_map (C++11) | No | Yes | No | Constant |
| std::multiset | Yes | No | Yes | Logarithmic |
| std::unordered\_multiset (C++11) | No | No | Yes | Constant |
| std::multimap | Yes | Yes | Yes | Logarithmic |
| std::unordered\_multimap (C++11) | No | Yes | Yes | Constant |

### Ordered Associative Containers

These containers store and sort elements in an **increasing order by the key**. In case the keys are of string type, they are sorted lexicographically.

The access time to the key and therefore to the value is logarithmic.

The most commonly-used ordered associative container in C++ is std::map. For example:

std::map<int, string> int2Str{ {1, "one"}, {2, "two"},

                               {3, "three"}, {4, "four"},

                               {5, "five"}, {6, "six"} };

### Unordered Associative Containers (C++11)

These containers (similar to [*hash tables*](https://www.youtube.com/watch?v=2Ti5yvumFTU&list=PL9IEJIKnBJjFiudyP6wSXmykrn67Ykqib&index=4&t=0s) in C ?) store elements NOT in order by the key. The key is mapped with the help of the hash function onto the bucket. You can find the key/value pair in the bucket.

There are some new terms which describe the characteristics of unordered associative containers:

* **Hash value**: The value you will get if you apply the hash function onto the key.
* **Collision**: If different keys are mapped to the same hash value, we will have a collision. The unordered associative containers have to deal with this situation.

The usage of the hash function has very important consequences for the unordered associative container:

* The keys have to support equal comparison to deal with collisions.
* The hash value of a key has to be available.

The execution of a hash function is a constant. Therefore, the access time to the keys of an unordered associative container is constant.

The most often used unordered associative container in C++11 is std::unordered\_map. For example:

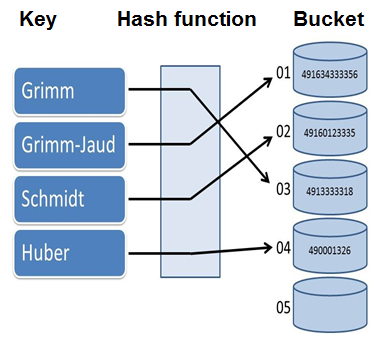
std::unordered\_map<string, int> str2Int{ {"Grimm", 491634333356},

                                         {"Grimm-Jaud", 49160123335},

                                         {"Schmidt", 4913333318},

                                         {"Huber", 490001326} };

The graphic below shows the mapping of the keys to their bucket using a hash function:



You might not know!

* The underlying structure of ordered associate containers are typically a *binary search tree*.
* **Ordered associate containers are generally *slower* than unordered associate containers** but certainly have their use if ordered access is necessary.
* **Selecting between ordered and unordered associative containers:**

<https://embeddedartistry.com/blog/2017/08/30/choosing-the-right-container-associative-containers/>

# 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**, such as an attempt to divide by zero. 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 parameter 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 type of this parameter is very important, since the type of the argument 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:

[Live Demo](http://tpcg.io/Nuo9hc)

#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) 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.

**2) Functions/methods can handle any exception they choose**

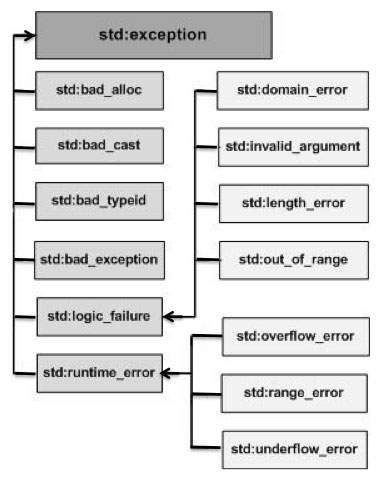
A function can throw many exceptions, but may choose to handle some of them. Other exceptions, thrown but not caught, can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

**3) 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. |

## 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:

[Live Demo](http://tpcg.io/FUdUJO)

#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>

# Assertions

## 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

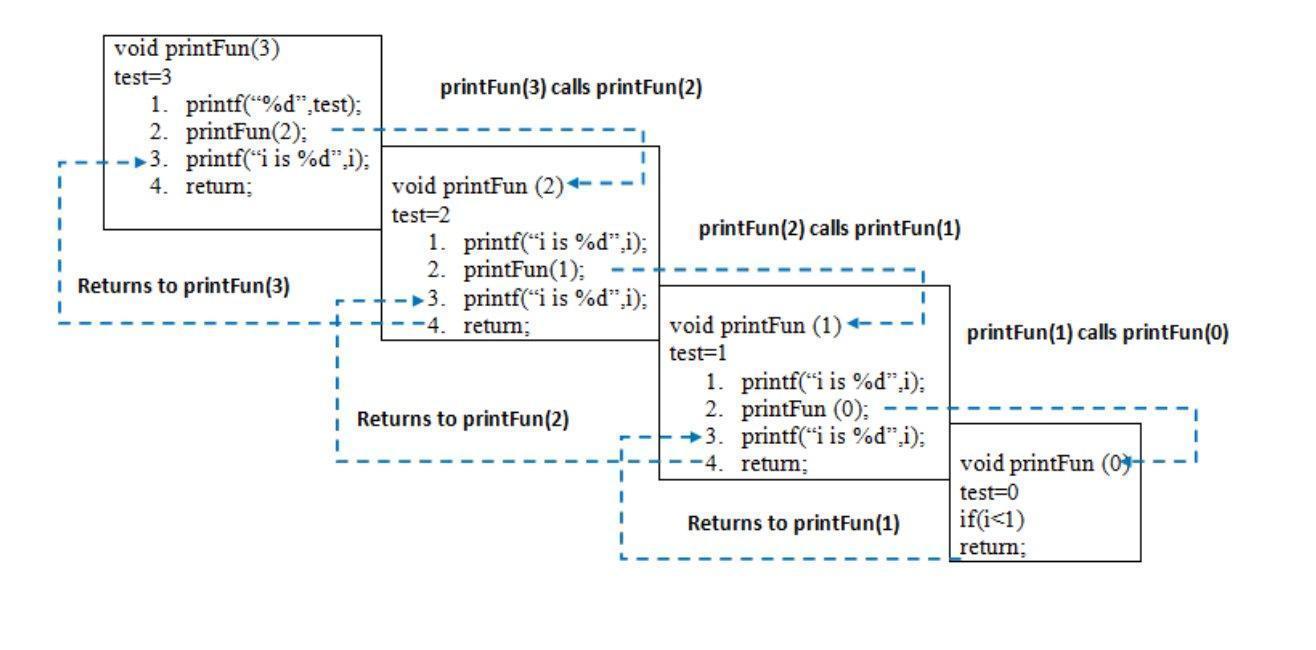
When printFun(3) is called from main(), memory is allocated to printFun(3) and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints ‘3’.

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 Pointers

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. Now 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 someFunction 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 = someFunction; // & removed

pf(5); // \* removed

**3)** Like normal pointers, we can have an array of function pointers.

**4)** Like normal data 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 fun() as parameter and calls the passed function during runtime.

#include <stdio.h>

void fun1() { printf("Fun1\n"); }

void fun2() { printf("Fun2\n"); }

void wrapper(void (\*fun)()) {

fun();

}

int main() {

wrapper(fun1);

wrapper(fun2);

return 0;

}

Output:

Fun1

Fun2

**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 algorith 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.

## Example

**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)() ) {

ptr();

}

int main() {

void (\*ptr)() = &A;

B(ptr);

// to make the code neater, we can replace two above statements by:

// B(A);

return 0;

}

Output:

I am A

**2.** This function shows you the most common use case of callback functions.

Many built-in functions in standard libraries require a callback as its argument. For example, the 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};

// void qsort (void\* base, size\_t num, size\_t size, int (\*com)(const void\*,const void\*));

qsort(arr, sizeof(arr)/sizeof(\*arr), sizeof(\*arr), cmp);

int i = 0;

while (i < sizeof(arr)/sizeof(\*arr))

{

printf("%d ", arr[i]);

i++;

}

return 0;

}

Output:

9 8 7 6 5 4 3 2 1

# 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:

class Functor

{

public:

int operator()(int a, int b) {

return a < b;

}

};

int main() {

Functor f;

int ans = f(1, 2); // This is similar to int ans = f.operator()(1,2)

}

The last line shows how you call a function object 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.

## Function Object vs 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. 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.
* *Templates* - Function pointers do not play too well 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 itr = vec.begin(), end = vec.end();

while(itr != end) {

if(pred(\*itr)) {

ret\_vec.push\_back(\*itr);

}

++itr;

}

}

// Using function object

template <typename FuncType>

std::vector<int> find\_matching\_numbers(std::vector<int> vec, FuncType pred) {

std::vector<int> ret\_vec;

std::vector<int>::iterator itr = vec.begin(), end = vec.end();

while(itr != end) {

if(pred(\*itr)) {

ret\_vec.push\_back(\*itr);

}

++itr;

}

}

* *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) {}

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 plainFunc() {

std::cout << "I'm a plain 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;

// Plain function

f.push\_back(plainFunc);

// 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 plain 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

# 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.

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; CBase\* pB;

CDerived d; CDerived\* pD;

pB = dynamic\_cast<CBase\*>(&d); // right: derived-to-base

pD = dynamic\_cast<CDerived\*>(&b);// wrong: base-to-derived. Only right when polymorphic

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);

if (pD==0) cout << "Null pointer on first type-cast" << endl;

pD = dynamic\_cast<CDerived\*>(pBb);

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

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);

This would be valid, although pB would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

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 = reinterpret\_cast<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.

# Overloading (Operator and Function)

C++ allows you to **specify more than one definition for a function name or an operator** **in the same scope**, which is called function overloading and operator overloading respectively

Both declarations have **different arguments and obviously different definition** (implementation). You cannot overload function declarations that differ only by return type.

When you call an overloaded function or operator, 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**.

## Function Overloading

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

[Live Demo](http://tpcg.io/cR3W4M)

#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;

}

When the above code is compiled and executed, it produces the following result:

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

## Operator Overloading

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 **functions with special names**: the keyword "***operator***" followed by the symbol for the operator being defined. **L**ike any other function, **an overloaded operator has a return type and a parameter list**.

Box operator+(const Box&)

The above statement declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class, then we would have to pass two arguments for each operand as follows:

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

Following is the example to show the concept of operator overloading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** pointer as explained below:

[Live Demo](http://tpcg.io/4Alfgt)

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

void setLength( double len ) {

length = len;

}

void setBreadth( double bre ) {

breadth = bre;

}

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.breadth = this->breadth + b.breadth;

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

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth 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.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(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;

}

When the above code is compiled and executed, it produces the following result:

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:

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

## Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

|  |  |
| --- | --- |
| **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) |

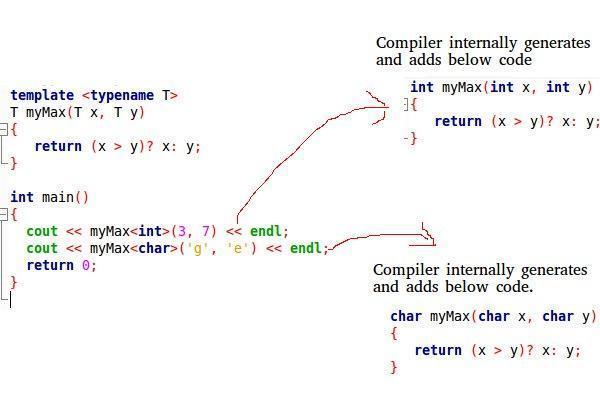
# Templates

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.



## Different Types of Template

### Function Templates

Using a template, we can write a generic function for different data types. Typical examples are sort(), max(), min(), printArray().

#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

Below is the program to implement [Bubble Sort](http://www.geeksforgeeks.org/bubble-sort/) using templates in C++:

#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.

Following is a simple example of template Array class.

#include <iostream>

using namespace std;

template <typename T>

class Array

{

    T \*ptr;

    int size;

public:

    Array(T arr[], int s);

    void print();

};

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

## Q&A

**Can there be more than one argument to templates?**

Yes, like normal parameters, we can pass more than one data types as arguments to templates. For example:

#include<iostream>

using namespace std;

template<class T, class U>

class A

{

    T x;

    U y;

public:

    A() {    cout << "Constructor called" << endl;   }

};

int main()

{

   A<char, char> a;

   A<int, double> b;

   return 0;

}

Output:

Constructor called

Constructor called

**Can we specify default value for template arguments?**

Yes, like normal parameters, we can specify default arguments to templates. For example:

#include<iostream>

using namespace std;

template<class T, class 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;

}

Output:

Constructor called

**What happens when there is static member in a template class/function?**

Each instance of a template contains its own static variable. See [Templates and Static variables](https://www.geeksforgeeks.org/templates-and-static-variables-in-c/) for more details.

**What is template specialization?**

Template specialization allows us to have different code for a particular data type. See [Template Specialization](https://www.geeksforgeeks.org/template-specialization-c/) for more details.

**Can we pass nontype parameters to templates?**

We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template. The important thing to note about non-type parameters is, they must be const. The compiler must know the value of non-type parameters at compile time. Because compiler needs to create functions/classes for a specified non-type value at compile time. In below program, if we replace 10000 or 25 with a variable, we get a compiler error. Please see [this](https://ide.geeksforgeeks.org/mgvysu).

// A C++ program to demonstrate working of non-type

// parameters to templates in C++.

#include <iostream>

using namespace std;

template <class T, int max>

int arrMin(T arr[], int n)

{

   int m = max;

   for (int i = 0; i < n; i++) {

      if (arr[i] < m) {

         m = arr[i];

}

   }

   return m;

}

int main()

{

   int arr1[]  = {10, 20, 15, 12};

   int n1 = sizeof(arr1)/sizeof(arr1[0]);

   char arr2[] = {1, 2, 3};

   int n2 = sizeof(arr2)/sizeof(arr2[0]);

   // Second template parameter to arrMin must be a constant

   cout << arrMin<int, 10000>(arr1, n1) << endl;

   cout << arrMin<char, 256>(arr2, n2);

   return 0;

}

Output:

10

1

**What is template metaprogramming?**

See [Template Metaprogramming](https://www.geeksforgeeks.org/template-metaprogramming-in-c/)

# 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 Variable

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 Return (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 and C++, 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 that big of a 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). What do I mean?

int x = 3;

decltype(x) y = x; // same as auto y = x;

Of couse, this is not a useful use of decltpe.

Now, you’ll find it useful when combining it with typedef:

std::vector<int> vec;

typedef decltype(vec.begin()) IT;

IT another\_iterator;

Or with template:

decltype(builder.makeObject());

This would give us the type returned from makeObject, allowing us to specify the return value from makeAndProcessObject. We can combine this with the *trailing return type* to produce this method:

template <typename Derive>

auto makeAndProcessObject(const Derive& builder) -> decltype(builder.makeObject()) {

auto val = builder.makeObject();

// do stuff with val

return val;

}

Note that the below lines don't work:

template <typename Derive>

decltype(builder.makeObject()) makeAndProcessObject(const Derive& builder) {

...

}

# 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 << ' '; // directly get element without an index

}

std::cout << '\n';

// Iterate over vector

std::vector<int> vec = {0, 1, 2, 3, 4, 5}; // uniform initializaton in C++11

for (auto i : vec) { // we can clearly 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> MAP({{1, 1}, {2, 2}, {3, 3}}); // uniform initializaton in C++11

for (auto i : MAP) {

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;

}

Outout:

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).

# 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");

# Smart Pointers (C++11)

## Advatages of Smart Pointers Over Raw Pointers

**No Explicit Delete**

Consider the following simple C++ code with raw pointers:

MyClass \*ptr = new MyClass();

ptr->doSomething();

// We must write this line to avoid memory leak

delete ptr;

Using smart pointers, we can make pointers to work in way that 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).

For how to implement a simple smart pointer, check [here](https://www.geeksforgeeks.org/smart-pointers-cpp/).

**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 SomeMethod()

{

ClassA \*a = new ClassA;

SomeOtherMethods(); // 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. We should replace auto\_ptr with unique\_ptr or scoped\_ptr from Boost if possible.

void SomeMethod()

{

std::auto\_ptr<ClassA> a(new ClassA);

SomeOtherMethods(); // 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. That is why the use of raw pointers should be avoided and appropriate smart pointers should be used instead.

**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, for example shared\_ptr from Boost. They use atomic operations for increasing/decreasing a reference counter, so it is thread safe.

## Standard Library Smart Pointers (Header File: <memory>)

### std::auto\_ptr

deprecated

Why? Check [Common Mistake #7: Using "auto\_ptr" (Incorrectly)](https://www.toptal.com/c-plus-plus/top-10-common-c-plus-plus-developer-mistakes)

### 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.

Replaces auto\_ptr, which is deprecated. 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:**

std::unique\_ptr<Type> p(new Type);

For example:

std::unique\_ptr<int> p1(new int);

std::unique\_ptr<int[]> p2(new int[50]);

std::unique\_ptr<Object> p3(new Object("Lamp"));

In C++14, it is also possible to construct unique\_ptrs 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).

**Usage:**

Consider this:

void compute()

{

std::unique\_ptr<int[]> data = std::make\_unique<int[]>(1024);

// do some computation on your data...

} // 'data' goes out of scope here: it is automatically destroyed

int main()

{

compute();

}

**Interface:**

* release(): To release ownership of the pointer.

**Note:**

unique\_ptr is very jealous of the dynamic object it holds: you **can't have multiple references** to its dynamic data. For example:

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

}

Technically this happens because a unique\_ptr doesn't have a copy constructor: it might be obvious to you if you are familiar with [move semantics](https://www.internalpointers.com/post/c-rvalue-references-and-move-semantics-beginners).

### std::shared\_ptr

A shared\_ptr owns the object it points to but, unlike unique\_ptr, it 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:**

std::shared\_ptr<Type> p(new Type);

For example:

std::shared\_ptr<int> p1(new int);

std::shared\_ptr<Object> p2(new Object("Lamp"));

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");

**Usage:**

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 or references with the method use\_count(). Consider this:

void compute()

{

std::shared\_ptr<int> ptr1 = std::make\_shared<int>(100); // ptr.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.

**Interface:**

shared\_ptr provides dereferencing operators \* and -> like a normal pointer provides. Besides, it provides some important interfaces:

* get(): To directly access the raw pointer pointing to the resource associated with the shared\_ptr .
* reset(): To release ownership of the pointer the destroy the resouce. This is useful when you want to free the memory owned by the smart pointer before the smart pointer goes out of scope.
* unique: To know whether the resource is managed by only this shared\_ptr instance.
* operator bool: To check whether the shared\_ptr owns a memory block or not. Can be used with an if condition.

**Note:** Circular references 🡪 Memory leaks

struct Player {

std::shared\_ptr<Player> companion;

~Player() { std::cout << "Destructor"; } // 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***. At the beginning, I create jasmine and albert to store dynamically-created objects: let's call this dynamic data jasmine-data and albert-data to make things clearer. Then, in (1) I 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.

### 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. Required in cases to **break circular references** between shared\_ptr instances.

**Syntax:**

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.

**Usage:**

A weak\_ptr is a sort of inspector on the shared\_ptr it depends on. You have to convert it to a shared\_ptr first with the lock() method if you really want to work with the actual object:

std::shared\_ptr<int> p\_shared = std::make\_shared<int>(100);

std::weak\_ptr<int> p\_weak(p\_shared);

...

std::shared\_ptr<int> p\_shared\_orig = p\_weak.lock();

Of course p\_shared\_orig might be null in case p\_shared got deleted somewhere else.

**Problem solver:**

A weak\_ptr makes the problem of dangling pointers (pointers that point to already deleted data) easy to solve. It provides the expired() method which checks whether the referenced object was already deleted.

In addition, a weak\_ptr breaks a circular reference. Let's go back to the Player example above and change the member variable from shared\_ptr<Player> companion to weak\_ptr<Player> companion to dissolve the entangled ownership. The actual dynamically-allocated data stays in the main body, while each Player has now a weak reference to it. Run the code with the change and you will see how the destructor gets called twice, correctly.

## 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. First of all, they let us forget about the new keyword. When working with smart pointers, we want to get rid of the new/delete combo, right? Secondly, 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 correctly specified in the C++ standard: their destructors will be triggered and the memory freed.

## Rules When Using Smart Pointers

* Always create smart pointers on a separate line of code, never in a parameter list, so that a subtle resource leak won't occur due to certain parameter list allocation rules.
* **Issues with arrays**

When a smart pointer exits from a scope, it will call a delete operator without [] brackets which results in incorrect behaviors. If using of a smart pointer is required for an array, it is possible to use scoped\_array or shared\_array from Boost or a unique\_ptr<T[]> specialization.

Another way is to create a workaround by using a custom deleter. One of the many shared\_ptr constructors takes a lambda as second parameter, where you manually delete the object it owns. For example:

std::shared\_ptr<int[]> p2(new int[16], [] (int\* i) {

delete[] i; // Custom delete

});

Unfortunately there's no way to do this when using std::make\_shared.

# Move Sematics (C++11)

# 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.

# 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();

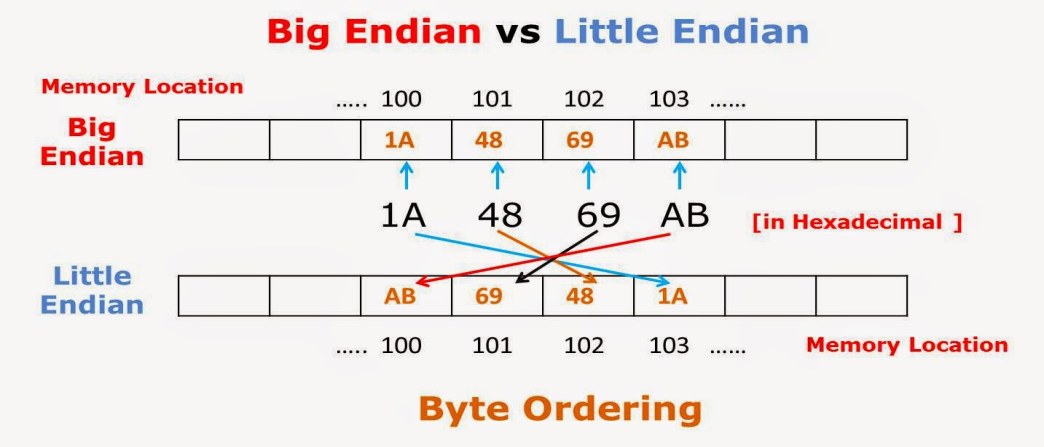
[[deprecated("Use new\_method instead")]]

void legacy\_method();

# Byte Ordering – Big Endian vs Little Endian

## What Are Big Endian and Little Endian?

<https://www.youtube.com/watch?v=seZLUbgbB7Y>

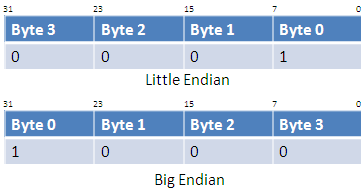


Little and big endian are two ways of storing multi-byte data-types (int, float, etc).

* 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 offfets 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 takes 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 forrmat (e.g, JPEG files) are stored in big endian format.

# DLL

**Great short video series about building and using libraries in C:**

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFn6zQQkJ2e8vxCVxhl2yuD>

## What Is DLL?

A DLL (Dynamic Link Library) is an **executable file** (.dll extension in Windows). It acts as a **shared library of codes, data and resources.**

Each DLL is a separate file, meaning that it’s not copied into the executable file (.exe) of the application. Instead, **the EXE will link to the DLL file when it’s running**. Codes and resources in a DLL file, as a result, will be compiled and deployed when the file is linked to the application.

The OS can load the DLL into the executable's memory space when the executable file is loaded, or on demand **at runtime**.

## Advantages of DLL

The biggest advantage of DLLs is that they make it easy to share functions and resources across multiple executable files. Multiple applications can also access the contents of a single copy of a DLL in memory at the same time.

**1. Uses Fewer Resources**

DLL files don't get loaded into the RAM together with the main program; they **don't occupy space unless required**. When a DLL file is needed, it is loaded and run. For example, as long as a user of MS Word is editing a document, the printer DLL file is not required in RAM. It is only loaded if the user decides to print the document.

**2. Promotes Modular Architecture**

A DLL helps promote developing modular programs. It helps you **develop large programs** that require multiple language versions or a program that requires modular architecture.

**3. Aid Easy Deployment and Installation**

When a function within a DLL needs an update or a fix, the deployment and installation of the DLL **does not require the program to be relinked with the DLL**. Additionally, if multiple programs use the same DLL, then all of them get benefited from the update or the fix. This issue may occur more frequently when you use a third-party DLL that is regularly updated or fixed.

Applications and DLLs can link to other DLLs automatically, if the DLL linkage is specified in **the IMPORTS section of the module definition file as a part of the compile**. Else, you can **explicitly load them using the Windows’ LoadLibrary function**.

## Export and Import DLLs in Visual Studio

<https://docs.microsoft.com/en-us/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp?view=vs-2017>

## Dynamic-Link Library Functions

<https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-functions>

<https://blog.benoitblanchon.fr/getprocaddress-like-a-boss/>

JetDrive example:

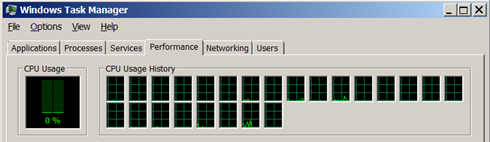
* *PrintMonitorGUI\_GetCallbacks*
* *Common/CCommonCBRProc\_OP\_IPDS.cpp*
* *Common/PrintMonitorGUI\_Callbacks.h*
* *PrintMonitorGUI/Plugin\_IF.h (and .cpp)*
* *PrintMonitorGUI/PrintMonitorGUI.h (and .cpp)*
* *PrintMonitorGUI/PrintMonitorGUI\_OP.h (and .cpp)*

*Plugin\_IF* is the interface for communication between different plugins. Its main implementation is about importing and exporting API from/to plugins.

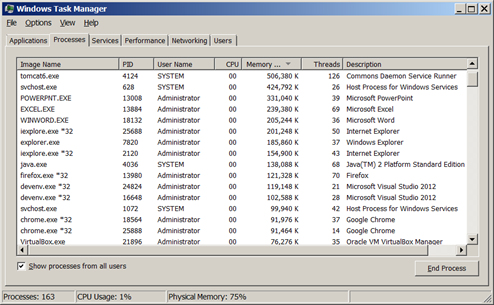
# Multi-Processing

**Modern CPUs provide multiple processing cores that allow multiple applications to run concurrently**.

For example, Windows Task Manager shows 24 CPU cores (independent processing units = each CPU core is an independent processor) on a workstation:



From the same computer, we can see many applications running at the same time:

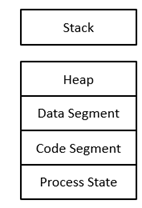


**Each program runs as a ‘*p*rocess*’* and is assigned a Process ID (PID)**.

Each process in memory shares some common traits:

* Firstly, a process needs to keep track of the state of the CPU processor it’s running under (**Process State**). This includes any program counters, register values, stack pointer (see below) and file/resource handles.
* Secondly, the process will include the **Code Segment** which is the actual programming code (machine language) it’s running.
* Thirdly, the **Data Segment** which holds statically declared data structures such as strings and other data defined by the programmer.
* Next, the **Heap** is where any dynamically allocated memory will be placed. For example, if your program creates five new objects, these will be stored in the Heap when the program is running.
* Finally, the **Stack** is a memory area that holds parameter values as they are passed to function or method calls.

Below is a basic diagram showing how these sections of memory are divided up for each process:



As long as one process is operating on its own set of data (such as single-processed, single-threaded programs), this model works out very well.

As said, modern operating systems like Windows and Linux allow us to run multiple programs like this at the same time. However, the limitation here is that **such programs themselves can only work on one thing at a time**. So if your program is processing a data file, it basically has to start at the beginning and go through the entire file to the end. While it is processing the data, the single-threaded program cannot do anything else.

On the other hand, **if we want our program to take advantage of multiple CPUs (processors), we need a way of allowing our program to carry out some work in parallel, which would require the program to share the data**.

## Image Processing Example

One of the classic examples of this type of parallel work can be found with image processing. For example, applying a filter to an image requires processing (doing some math on) each pixel. With one process, it will take some time to visit each pixel and process it. For example:

In the meantime, the program cannot do anything else until it is all done with running the image filter.

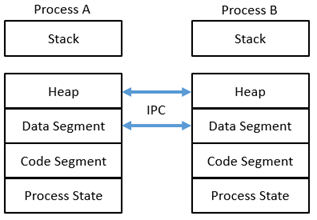
However, images can be segmented into a collection of “tiles” and each tile can be processed independently. This can be accomplished if the work to be done can operate on each pixel independently. For example, if we want our picture to be lighter or brighter in color, then processing pixels in the upper left hand corner of the picture will have nothing to do with processing the pixels in the lower right-hand corner of the picture.

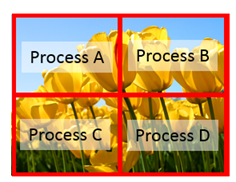
Some ways to describe this type of data processing are parallel processing or **multi-threaded** processing of data. The “parallel” term has been used for some time to describe situations when the software program can work on different portions of the data at the same time.

In the next section, I will describe two ways in which we can carry out this independent processing of data.

## How Does Multi-Processing Work?

In this section, I will introduce two ways in which we can process data in parallel.

One way to work on data in parallel is to run multiple processes and then use an Inter-Process Communications (**IPC**) method to share the data. Some IPC methods include **Sockets**, **Pipes**, **Remote** **Procedure** **Calls**, **Shared** **Memory** and various forms of Message Passing Interfaces (**MPI**). For example, Process A might run and open up our data file to be processed, and then pass some of that data to Process B to split up the work as demonstrated below:

So we might have four processes; each work on a different portion of the image. The processes would need to communicate with one another and pass the image data around to work on it.

The good news is that creating additional processes is something that all modern CPUs and operating systems support very well. However, creating a completely new process each time we want to share the data will have a lot of overhead since we need to make a completely new process state, code segment, data segment and stack. Data on the Heap is shared but must be transmitted between processes using IPC. The bottom line here is that **using multiple processes to work on the same set of data is going to be inefficient**.

# Multi-Threading

**Great short video series about programming with threads in C:**

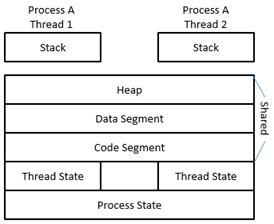
<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFZxuqyJ9JqVYmuFZHr7CFM>

**Great short video series about programming with processes in C:**

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFNNfpY6fHjVzAwtgRYjhPw>

## How Does Multi-Threading Work?

A thread is a part of our software (Code Segment) that can run independently of the rest of the process. For example, one process with two threads will share the Code Segment, Data Segment and Heap. Much of the Process State can be shared as well; however, each thread will need some thread-specific state information held aside. In addition, each thread needs its own Stack space to keep track of the function and method calls.

Because there is less work to be done to create a new thread (as compared to creating a new process), we can say typically **using multiple threads will result in less overhead and more efficient processing**. 

Going back to our image-processing example, our program might launch 4 threads to process different regions of the image. Since the image data is shared on the memory Heap, the threads do not need to pass anything between them using IPC.

## Multi-Threading APIs

Most modern OS support threads at the OS level. Within C and C++, there are various threading libraries you can choose from:

* POSIX Threads, also called pthreads (C and C++)
* Boost Threads Library (C++)
* C++11 Standard Library Threads (C++11)
* Win32 Library Threads (C and C++)

### C++11 Standard Thread Library

To use C++11 thread library, you will need an updated C++ compiler that supports the C++11 standard. On MS Windows, you may use Microsoft Visual Studio 12 and Visual Studio 13. On Linux, you will need an updated version of the GNU C++ Compiler such as version 4.8.1 or later.

$ g++ --version

g++ (Ubuntu 4.8.2-19ubuntu1) 4.8.2

To compile with GNU C++ compiler,

* On Linux, use the -std=c++11 and -pthread command line.
* On MacOS, need the C/C++ Language development tools that come with XCode:

More docs need to copy: <http://holowczak.com/introduction-to-multithreading-in-cplusplus/6/>

<https://www.youtube.com/watch?v=LL8wkskDlbs>

#### Example 1: Worker Function Without Parameters

Create and build the code below using Visual Studio 12 or higher:

#include <stdafx.h>

#include <iostream>

#include <thread>

#include <chrono>

void worker\_functionA(void)

{

int loop = 0;

// Loop 10 times and print to the screen from 1 to 9

while (loop < 10) {

// Sleep for 1.33 seconds

std::this\_thread::sleep\_for(std::chrono::milliseconds(1333));

std::cout << "Thread A Reporting: " << loop << std::endl;

loop++;

}

}

void worker\_functionB(void)

{

int loop = 0;

// Loop 10 times and print to the screen from 1 to 9

while (loop < 10) {

// Sleep for 2.22 seconds

std::this\_thread::sleep\_for(std::chrono::milliseconds(2222));

std::cout << "Thread B Reporting: " << loop << std::endl;

loop++;

}

}

int main()

{

char result;

// Launch two new threads. They will start executing immediately

std::thread worker\_threadA(worker\_functionA);

std::thread worker\_threadB(worker\_functionB);

// Pause the main thread

std::cout << "Press a key to finish" << std::endl;

std::cin >> result;

// Join up the two worker threads back to the main thread

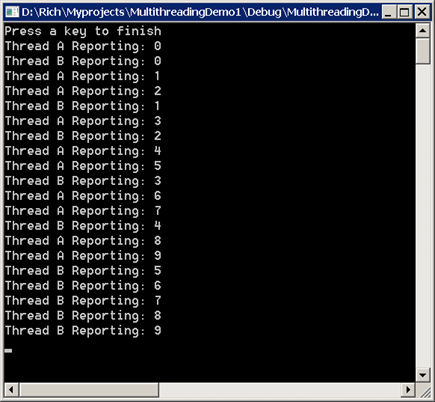
worker\_threadA.join();

worker\_threadB.join();

return 1;

}

Output:



So basically, we will have 3 threads: main thread, thread for functionA and thread for functionB.

#### Example 2: Worker Function With Parameters

One common feature we may want to take advantage of is to generalize the worker functions especially in cases where they are doing very **similar types of processing just on different subsets of the data or with slightly different operating parameters**. To accomplish this we can pass parameters to the thread when it is invoked.

In the C++11 standard, this is actually quite easy to do. When we call the constructor for std::thread we can easily pass along the necessary parameters.

For example:

#include <stdafx.h>

#include <iostream>

#include <thread>

#include <chrono>

void worker\_function(int thread\_number, int iterations, long delay)

{

int loop = 0;

// Loop some 'iterations' number of times

while (loop < iterations)

{

// Sleep for some time

std::this\_thread::sleep\_for(std::chrono::milliseconds(delay));

std::cout << "Thread " << thread\_number << " Reporting: " << loop << " with delay " << delay << std::endl;

loop++;

}

}

int main()

{

char result;

// Launch two new threads. They will start executing immediately

// worker\_function is a generic worker that will take in two parameters:

// number of iterations and sleep time

std::thread worker\_thread1(worker\_function, 1, 10, 1555);

std::thread worker\_thread2(worker\_function, 2, 10, 2222);

// Pause the main thread

std::cout << "Press a key to finish" << std::endl;

std::cin >> result;

// Join up the two worker threads to the main thread

worker\_thread1.join();

worker\_thread2.join();

// Return success

return 1;

}

Output:

Press a key to finish

Thread 1 Reporting: 0 with delay 1555

Thread 2 Reporting: 0 with delay 2222

Thread 1 Reporting: 1 with delay 1555

Thread 2 Reporting: 1 with delay 2222

Thread 1 Reporting: 2 with delay 1555

Thread 1 Reporting: 3 with delay 1555

Thread 2 Reporting: 2 with delay 2222

Thread 1 Reporting: 4 with delay 1555

Thread 2 Reporting: 3 with delay 2222

Thread 1 Reporting: 5 with delay 1555

Thread 1 Reporting: 6 with delay 1555

Thread 2 Reporting: 4 with delay 2222

Thread 1 Reporting: 7 with delay 1555

Thread 2 Reporting: 5 with delay 2222

Thread 1 Reporting: 8 with delay 1555

Thread 2 Reporting: 6 with delay 2222

Thread 1 Reporting: 9 with delay 1555

Thread 2 Reporting: 7 with delay 2222

Thread 2 Reporting: 8 with delay 2222

Thread 2 Reporting: 9 with delay 2222

#### Example 3: Using a Mutex to Ensure Properly-Shared Resources

One of the **issues we will come up against when running multiple threads is the possibility of multiple threads access the same data at the same time**. If all of our threads are only reading the data, then we generally do not have much to worry about. However if one or more threads is writing some data (such as modifying the value of a variable) while other threads may be reading the data, then we face a potential data corruption problem. We can extend the same logic to any resource including memory, files and any other system devices.

One such device is the console itself where std::cout is directed. For this exercise, we will want to protect std::cout to make sure our two threads do not output text at the same time, otherwise the output lines will be mixed up.

To solve this issue, we employ a **Mutex. Each mutex acts as a barrier, which takes on one of two states: Locked and Unlocked**. The working principle is as follows:

* When our thread worker function reaches a critical section where it will work with some shared resource, our thread will obtain the lock by calling the **.lock()** method.
* Once the thread is done working on the critical section, it will call the **.unlock()** method.

Once the lock has been acquired, any other thread attempting to get the lock will be held in a **wait state until the lock is freed up**.

#include <stdafx.h>

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

// Mutex to protect the console output (std::cout)

std::mutex mtConsole;

void worker\_function(int thread\_number, int iterations, long delay)

{

int loop = 0;

// Loop some 'iterations' number of times

while (loop < iterations)

{

// Sleep for some time

std::this\_thread::sleep\_for(std::chrono::milliseconds(delay));

// Get the lock on the console

mtConsole.lock();

std::cout << "Thread " << thread\_number << " Reporting: "

<< loop << " with delay " << delay << std::endl;

// Remove the lock on the console

mtConsole.unlock();

loop++;

}

}

int main()

{

char result;

// Launch two new threads. They will start executing immediately

std::thread worker\_thread1(worker\_function, 1, 10, 2000);

std::thread worker\_thread2(worker\_function, 2, 10, 2000);

// Pause the main thread

std::cout << "Press a key to finish" << std::endl;

std::cin >> result;

// Join up the two worker threads to the main thread

worker\_thread1.join();

worker\_thread2.join();

// Return success

return 1;

}

Output:

Same output as example 2

#### Example 4: Creating a Reader/Writer Thread Pair

There are a number of application scenarios where one thread will populate objects with data while a second (or multiple other) thread reads this data. Such design patterns are called “Produce/Consumer” or “Publish/Subscribe” models.

For this example, we are attempting to simulate the dynamic arrival of market data such as stock prices from an exchange. Such data arrives randomly from a market data provider. There may be period with no updates and periods where there are a high frequency of updates. We simulate this in the data\_writer thread by imposing a random delay in between data “arrivals” which are then loaded into a vector (dynamic array).

We will have one thread that “writes” data into a dynamic array. A second thread will read data from this same dynamic array. To protect the dynamic array from inconsistent reads, we will use a Mutex with .lock and .unlock methods around each use of the array to avoid data corruption.

The data\_reader thread will then iterate over the vector and calculate an average price based on all of the prices that have been received so far. This activity will happen on a fixed time interval.

#include "stdafx.h"

#include <iostream>

#include <thread>

#include <chrono>

#include <mutex>

#include <vector>

// Mutex to protect our price array

std::mutex mtPriceArray;

// Create an array to hold our prices

std::vector<double> dblPriceArray;

void data\_writer()

{

// Write our market data into dblPriceArray

while (1) {

// random delay

// Sleep for some random time up to 5 seconds

std::this\_thread::sleep\_for(std::chrono::milliseconds(rand() % 5000));

// Lock the price array

mtPriceArray.lock();

// Create a price and load it into the array

// In a real application we would be pulling this price from our market data source

dblPriceArray.push\_back(20.0 \* (1 + (rand() % 100) / 1000.0));

// Unlock the price array

mtPriceArray.unlock();

}

}

void data\_reader(int delay)

{

unsigned int loop = 0;

double dblSum = 0;

double dblAverage = 0;

while (1) {

// Get the lock on the price array

mtPriceArray.lock();

if (dblPriceArray.size() > 0) {

// Calculate the average

dblSum = 0.0;

for (loop = 0; loop < dblPriceArray.size(); loop++) {

dblSum += dblPriceArray[loop];

}

dblAverage = dblSum / dblPriceArray.size();

std::cout << "Latest Price: " << dblPriceArray[dblPriceArray.size() - 1];

std::cout << " Average: " << dblAverage << std::endl;

}

// Unlock the price array vector

mtPriceArray.unlock();

// Sleep for some time

std::this\_thread::sleep\_for(std::chrono::milliseconds(delay));

} // end while

}

int main()

{

char result;

// Launch two new threads. They will start executing immediately

std::thread writer\_thread(data\_writer);

std::thread reader\_thread(data\_reader, 2000);

// Pause the main thread

std::cout << "Press a key to finish" << std::endl;

std::cin >> result;

// Join up the two worker threads to the main thread

writer\_thread.join();

reader\_thread.join();

// Return success

return 1;

}

Output:

Press a key to finish

Latest Price: 21.34 Average: 21.34

Latest Price: 20 Average: 20.67

Latest Price: 20.48 Average: 20.6067

Latest Price: 21.16 Average: 20.745

Latest Price: 20.9 Average: 20.86

Latest Price: 20.9 Average: 20.86

Latest Price: 20.54 Average: 20.8143

Latest Price: 20.54 Average: 20.8143

Latest Price: 21.82 Average: 20.94

Latest Price: 21.82 Average: 20.94

### Win32 Thread Libraries

<https://hackernoon.com/need-faster-code-try-multithreading-5dc30c83837c>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexa>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject>

<https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-releasemutex>

<https://www.tenouk.com/ModuleAA.html>

JetDrive example: ModeChange\Ini\_Template.h (and .cpp)

# Network

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjH_zM5LnovnoaKlXML5qh17>