

CPP Basics

Thursday, March 31, 2022 6:29 PM

- C++ is a middle-level language, as it encapsulates both high and low level language features.

Object-Oriented Programming (OOPs)

C++ supports the object-oriented programming, the four major pillar of object-oriented programming (OOPs) used in C++ are;

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

C++ Program

In this tutorial, all C++ programs are given with C++ compiler so that you can easily change the C++ program code.

File: main.cpp

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello C++ Programming";
    return 0;
}
```

#include<iostream.h> includes the **standard input output** library functions. It provides **cin** and **cout** methods for reading from input and writing to output respectively.

#include <conio.h> includes the **console input output** library functions. The **getch()** function is defined in **conio.h** file.

void main() The **main()** function is the entry point of every program in C++ language. The **void** keyword specifies that it returns no value.

cout << "Welcome to C++ Programming." is used to print the data **"Welcome to C++ Programming."** on the console.

- Stream is the sequence of bytes or flow of data. It makes the performance fast.

Header File	Function and Description
<iostream>	It is used to define the cout , cin and cerr objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as setprecision and setw .
<fstream>	It is used to declare services for user-controlled file processing.

```
#include <iostream>
using namespace std;
int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "Your age is: " << age << endl;
}
```

Output:

```
Enter your age: 22
Your age is: 22
```

- The endl is a predefined object of ostream class. It is used to insert a new line characters and flushes the stream.

```
#include <iostream>
using namespace std;
int main() {
    cout << "C++ Tutorial";
    cout << " Javatpoint" << endl;
    cout << "End of line" << endl;
}
```

Output:

```
C++ Tutorial Javatpoint
End of line
```

- A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

```
int x=5,b=10; //declaring 2 variable of integer type
float f=30.8;
char c='A';
```

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

```
int a;
int _ab;
int a30;
```

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

Basic Datatypes:

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 32,767
int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 32,767
short int	2 byte	-32,768 to 32,767
signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 32,767
long int	4 byte	
signed long int	4 byte	
unsigned long int	4 byte	
float	4 byte	

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. **A list of 32 Keywords in C++ Language which are also available in C language are given below.**

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

A list of 30 Keywords in C++ Language which are not available in C language are given below.

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	new	static_cast	false	catch
operator	template	friend	private	class
this	inline	public	throw	const_cast
delete	mutable	protected	true	try
typeid	typename	using	virtual	wchar_t

	Operator	Type
Binary Operator	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&, , !	Logical Operators
	&, , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator	→ ++, --	Unary Operator
Ternary Operator	→ ?:	Ternary or Conditional Operator

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Right to left
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Right to left
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Right to left
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

- C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer.

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

- **Constants**
- **Variables**
- **Functions**
- **Labels**
- **Defined data types**

Some naming rules are common in both C and C++. They are as follows:

- Only alphabetic characters, digits, and underscores are allowed.
- The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
- In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
- A declared keyword cannot be used as a variable name.
- The major difference between C and C++ is the limit on the length of the name of the variable. ANSI C considers only the first 32 characters in a name while ANSI C++ imposes no limit on the length of the name.
- Keywords are the reserved words that have a special meaning to the compiler. They are reserved for a special purpose, which cannot be used as the identifiers. For example, 'for', 'break', 'while', 'if', 'else', etc. are the predefined words where predefined words are those words whose meaning is already known by the compiler. Whereas, the identifiers are the names which are defined by the programmer to the program elements such as variables, functions, arrays, objects, classes.

Identifiers	Keywords
Identifiers are the names defined by the programmer to the basic elements of a program.	Keywords are the reserved words whose meaning is known by the compiler.
It is used to identify the name of the variable.	It is used to specify the type of entity.
It can consist of letters, digits, and underscore.	It contains only letters.
It can use both lowercase and uppercase letters.	It uses only lowercase letters.
No special character can be used except the underscore.	It cannot contain any special character.
The starting letter of identifiers can be lowercase, uppercase or underscore.	It can be started only with the lowercase letter.
It can be classified as internal and external identifiers.	It cannot be further classified.
Examples are test, result, sum, power, etc.	Examples are 'for', 'if', 'else', 'break', etc.

If Else if and else example

```
#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
    if (num <0 || num >100)
    {
        cout<<"wrong number";
    }
    else if(num >= 0 && num < 50){
        cout<<"Fail";
    }
    else if (num >= 50 && num < 60)
    {
        cout<<"D Grade";
    }
    else if (num >= 60 && num < 70)
    {
        cout<<"C Grade";
    }
    else if (num >= 70 && num < 80)
    {
        cout<<"B Grade";
    }
    else if (num >= 80 && num < 90)
    {
        cout<<"A Grade";
    }
    else if (num >= 90 && num <= 100)
    {
        cout<<"A+ Grade";
    }
}
```

C++ Switch Example

```
#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
    switch (num)
    {
        case 10: cout<<"It is 10"; break;
        case 20: cout<<"It is 20"; break;
        case 30: cout<<"It is 30"; break;
        default: cout<<"Not 10, 20 or 30"; break;
    }
}
```

Output:

```
Enter a number:
10
It is 10
```

- Default case executes only if none of the cases matched.

For Loop

- If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

C++ For Loop Example

```
#include <iostream>
using namespace std;
int main() {
    for(int i=1;i<=10;i++){
        cout<<i <<"\n";
    }
}
```

Nested For loop

C++ Nested For Loop Example

Let's see a simple example of nested for loop in C++.

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

int main () {
    for(int i=1;i<=3;i++){
        for(int j=1;j<=3;j++){
            cout<<i<<" "<j<<"\n";
        }
    }
}
```

C++ Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. Let's see a simple example of infinite for loop in C++.

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

int main () {
    for (;)
    {
        cout<<"Infinitive For Loop";
    }
}
```

Output:

```
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
ctrl+c
```


C++ While Loop Example

Let's see a simple example of while loop to print table of 1.

```
#include <iostream>
using namespace std;
int main() {
    int i=1;
    while(i<=10)
    {
        cout<<i <<"\n";
        i++;
    }
}
```

C++ Infinitive While Loop Example:

We can also create infinite while loop by passing true as the test condition.

```
#include <iostream>
using namespace std;
int main () {
    while(true)
    {
        cout<<"Infinitive While Loop";
    }
}
```

Output:

```
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
ctrl+c
```

C++ do-while Loop Example

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    do{
        cout<<i<<"\n";
        i++;
    } while (i <= 10);
}
```

C++ Infinitive do-while Loop Example

```
#include <iostream>
using namespace std;
int main() {
    do{
        cout<<"Infinitive do-while Loop";
    } while(true);
}
```

Output:

```
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
ctrl+c
```

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop.

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 10; i++)
    {
        if (i == 5)
        {
            break;
        }
        cout<<i<<"\n";
    }
}
```

C++ Continue Statement Example

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=10;i++){
        if(i==5){
            continue;
        }
        cout<<i<<"\n";
    }
}
```

C++ Continue Statement with Inner Loop

C++ Continue Statement continues inner loop only if you use continue statement inside the inner loop.

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=3;i++){
        for(int j=1;j<=3;j++){
            if(i==2&&j==2){
                continue;
            }
            cout<<i<<" "<<j<<"\n";
        }
    }
}
```

C++ Single Line Comment

The single line comment starts with // (double slash). Let's see an example of single line comment in C++.

C++ Multi Line Comment

The C++ multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk (/ * */). Let's see an example of multi line comment in C++.

C++ Function Example

Let's see the simple example of C++ function.

```
#include <iostream>
using namespace std;
void func() {
    static int i=0; //static variable
    int j=0; //local variable
    i++;
    j++;
    cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
    func();
    func();
    func();
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

```

#include <iostream>
using namespace std;
void change(int data);
int main()
{
    int data = 3;
    change(data);
    cout << "Value of the data is: " << data << endl;
    return 0;
}
void change(int data)
{
    data = 5;
}

```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

```

#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
    int swap;
    swap=*x;
    *x=*y;
    *y=swap;
}
int main()
{
    int x=500, y=100;
    swap(&x, &y); // passing value to function
    cout<<"Value of x is: " << x << endl;
    cout<<"Value of y is: " << y << endl;
    return 0;
}

```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.

```
#include<iostream>
using namespace std;
int main()
{
    int factorial(int);
    int fact,value;
    cout<<"Enter any number: ";
    cin>>value;
    fact=factorial(value);
    cout<<"Factorial of a number is: "<<fact<<endl;
    return 0;
}
int factorial(int n)
{
    if(n<0)
        return(-1); /*Wrong value*/
    if(n==0)
        return(1); /*Terminating condition*/
    else
    {
        return(n*factorial(n-1));
    }
}
```

Output:

```
Enter any number: 5
Factorial of a number is: 120
```

C++ Storage Classes

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

- Register variables must be used only when we need quick accessing of the memory. Because it stores the data on RAM , which is very limited.
- Static variable is initialized only once and exists till the end of a program. It retains it's value between multiple function calls.
- Extern variable is visible to all the programs , it is used if two or more files are sharing the same variables or functions.

C++ Single Dimensional Array

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
    //traversing array
    for (int i = 0; i < 5; i++)
    {
        cout<<arr[i]<<"\n";
    }
}
```

Output:/p>

```
10
0
20
0
30
```

Iterating through foreach loop

```
#include <iostream>
using namespace std;
int main()
{
    int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
    //traversing array
    for (int i: arr)
    {
        cout<<i<<"\n";
    }
}
```

Output:

```
10
20
30
40
50
```

C++ Passing Array to Function Example: print array elements

Let's see an example of C++ function which prints the array elements.

```
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
    int arr1[5] = { 10, 20, 30, 40, 50 };
    int arr2[5] = { 5, 15, 25, 35, 45 };
    printArray(arr1); //passing array to function
    printArray(arr2);
}
void printArray(int arr[5])
{
    cout << "Printing array elements:" << endl;
    for (int i = 0; i < 5; i++)
    {
        cout<<arr[i]<<"\n";
    }
}
```


Output:

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

C++ Multidimensional Array Example

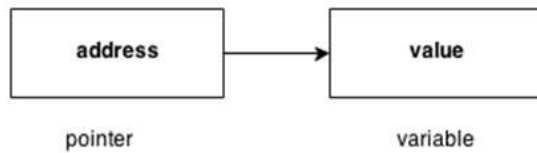
Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
#include <iostream>
using namespace std;
int main()
{
    int test[3][3]; //declaration of 2D array
    test[0][0]=5; //initialization
    test[0][1]=10;
    test[1][1]=15;
    test[1][2]=20;
    test[2][0]=30;
    test[2][2]=10;
    //traversal
    for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 3; ++j)
        {
            cout<< test[i][j]<< " ";
        }
        cout<< "\n"; //new line at each row
    }
    return 0;
}
```

```
int test[3][3] =
{
    {2, 5, 5},
    {4, 0, 3},
    {9, 1, 8} }; //declaration and initialization
```

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
#include <iostream>
using namespace std;
int main()
{
    int number=30;
    int * p;
    p=&number;//stores the address of number variable
    cout<<"Address of number variable is:"<<&number<<endl;
    cout<<"Address of p variable is:"<<p<<endl;
    cout<<"Value of p variable is:"<<*p<<endl;
    return 0;
}
```

Output:

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
    int a=20,b=10,*p1=&a,*p2=&b;
    cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
    *p1=*p1+*p2;
    *p2=*p1-*p2;
    *p1=*p1-*p2;
    cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
    return 0;
}
```

Output:

```
Before swap: *p1=20 *p2=10
After swap: *p1=10 *p2=20
```

The sizeof() is an operator that evaluates the size of data type, constants, variable. It is a compile-time operator as it returns the size of any variable or a constant at the compilation time.

The size, which is calculated by the sizeof() operator, is the amount of RAM occupied in the computer.

```
#include <iostream>

using namespace std;

class Base
{
    int a;
    int d;
    char ch;
};

int main()
{
    Base b;
    std::cout << "Size of class base is : "<<sizeof(b) << std::endl;
    return 0;
}
```

In the above code, the class has two integer variables, and one char variable. According to our calculation, the size of the class would be equal to 9 bytes (int+int+char), but this is wrong due to the concept of structure padding.

```
Size of class base is : 12

...Program finished with exit code 0
Press ENTER to exit console.
```

Pointer Memory Interpretation

```
#include <iostream>
using namespace std;
int main()
{
    int arr[]={10,20,30,40,50};
    std::cout << "Size of the array 'arr' is : " << sizeof(arr) << std::endl;
    return 0;
}
```

In the above program, we have declared an array of integer type which contains five elements. We have evaluated the size of the array by using **sizeof()** operator. According to our calculation, the size of the array should be 20 bytes as int data type occupies 4 bytes, and array contains 5 elements, so total memory space occupied by this array is $5 \times 4 = 20$ bytes. The same result has been shown by the **sizeof()** operator as we can observe in the following output.

Output

```
Size of the array 'arr' is : 20

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
using namespace std;
void fun(int arr[])
{
    std::cout << "Size of array is : " << sizeof(arr) << std::endl;
}
int main()
{
    int arr[]={10,20,30,40,50};
    fun(arr);
    return 0;
}
```

In the above program, we have tried to print the size of the array using the function. In this case, we have created an array of type integer, and we pass the '**arr**' to the function **fun()**. The **fun()** would return the size of the integer pointer, i.e., **int***, and the size of the **int*** is 8 bytes in the 64-bit operating system.

```
input
main.cpp:15:52: warning: 'sizeof' on array function parameter 'arr' will return size of 'int*' [-Wsizeof-array-argument]
main.cpp:13:18: note: declared here
Size of array is : 8

...Program finished with exit code 0
Press ENTER to exit console. □
```

- If the computer has 32bit operating system, then the size of the pointer would be 4 bytes. If the computer has 64-bit operating system, then the size of the pointer would be 8 bytes.
- We have declared two variables num1 and num2 of type int and double, respectively. The size of the int is 4 bytes, while the size of double is 8 bytes. The result would be the variable, which is of double type occupying 8 bytes.
- The first element of the array contains the address of the whole elements of the array. Using the first element address we find all the consecutive elements.

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr; // integer pointer declaration
    int marks[10]; // marks array declaration
    std::cout << "Enter the elements of an array : " << std::endl;
    for(int i=0;i<10;i++)
    {
        cin>>marks[i];
    }
    ptr=marks; // both marks and ptr pointing to the same element..
    std::cout << "The value of *ptr is : " << *ptr << std::endl;
    std::cout << "The value of *marks is : " << *marks << std::endl;
}
```

In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0]. When we try to print the values of *ptr and *marks, then it comes out to be same. Hence, it is proved that the array name stores the address of the first element of an array.

```
Enter the elements of an array :
1
2
3
4
5
6
7
8
9
10
The value of *ptr is :1
The value of *marks is :1
```

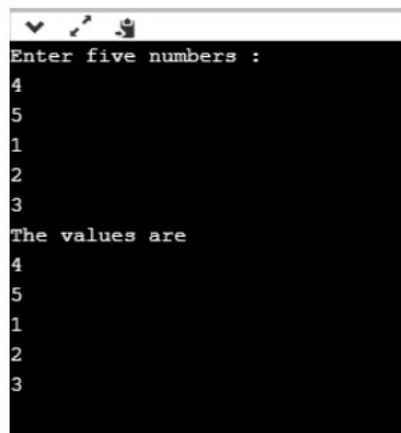
Array Of Pointers to Integers

```

#include <iostream>
using namespace std;
int main()
{
    int ptr1[5]; // integer array declaration
    int *ptr2[5]; // integer array of pointer declaration
    std::cout << "Enter five numbers : " << std::endl;
    for(int i=0;i<5;i++)
    {
        std::cin >> ptr1[i];
    }
    for(int i=0;i<5;i++)
    {
        ptr2[i]=&ptr1[i];
    }
    // printing the values of ptr1 array
    std::cout << "The values are" << std::endl;
    for(int i=0;i<5;i++)
    {
        std::cout << *ptr2[i] << std::endl;
    }
}

```

- In the above code, we declare an array of integer type and an array of integer pointers. We have defined the 'for' loop, which iterates through the elements of an array 'ptr1', and on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.



```

Enter five numbers :
4
5
1
2
3
The values are
4
5
1
2
3

```


Array of Pointer to Strings

An array of pointer to strings is an array of character pointers that holds the address of the first character of a string or we can say the base address of a string.

The following are the differences between an array of pointers to string and two-dimensional array of characters:

- An array of pointers to string is more efficient than the two-dimensional array of characters in case of memory consumption because an array of pointer to strings consumes less memory than the two-dimensional array of characters to store the strings.
- In an array of pointers, the manipulation of strings is comparatively easier than in the case of 2d array. We can also easily change the position of the strings by using the pointers.

```
char *names[5] = {"john",  
                 "Peter",  
                 "Marco",  
                 "Devin",  
                 "Ronan"};
```

In the above code, we declared an array of pointer names as 'names' of size 5. In the above case, we have done the initialization at the time of declaration, so we do not need to mention the size of the array of a pointer. The above code can be re-written as:

```
char *names[ ] = {"john",  
                 "Peter",  
                 "Marco",  
                 "Devin",  
                 "Ronan"};
```

In the above case, each element of the 'names' array is a string literal, and each string literal would hold the base address of the first character of a string. For example, names[0] contains the base address of "john", names[1] contains the base address of "Peter", and so on. It is not guaranteed that all the string literals will be stored in the contiguous memory location, but the characters of a string literal are stored in a contiguous memory location.

```

#include <iostream>
using namespace std;
int main()
{
    char *names[5] = {"john",
                      "Peter",
                      "Marco",
                      "Devin",
                      "Ronan"};
    for(int i=0;i<5;i++)
    {
        std::cout << names[i] << std::endl;
    }
    return 0;
}

```

In the above code, we have declared an array of char pointer holding 5 string literals, and the first character of each string is holding the base address of the string.

Output



```

john
Peter
Marco
Devin
Ronan

```

- A void pointer is a general-purpose pointer that can hold the address of any data type, but it is not associated with any data type.

Syntax of void pointer

```
void *ptr;
```

```

int *ptr; // integer pointer declaration
float a=10.2; // floating variable initialization
ptr= &a; // This statement throws an error.

```



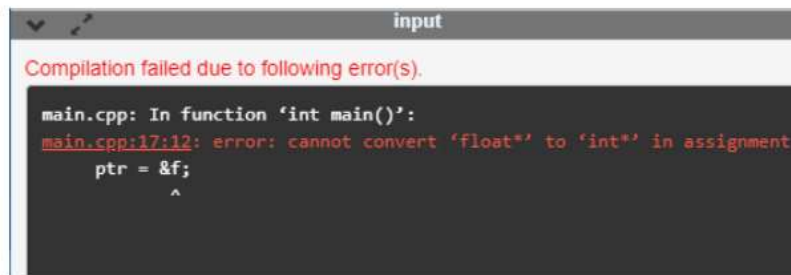
```

#include <iostream.h>
using namespace std;
int main()
{
    int *ptr;
    float f=10.3;
    ptr = &f; // error
    std::cout << "The value of *ptr is : " <<*ptr<< std::endl;
    return 0;
}

```

In the above program, we declare a pointer of integer type and variable of float type. An integer pointer variable cannot point to the float variable, but it can point to an only integer variable.

Output



```

input
Compilation failed due to following error(s).
main.cpp: In function 'int main()':
main.cpp:17:12: error: cannot convert 'float*' to 'int*' in assignment
    ptr = &f;
           ^

```

- To solve such errors C++ introduces void pointer, where we can declare a pointer of void type and can store address of pointer of any datatype.
- In C, we do not need to typecast a void pointer to any type of pointer while assigning. But such type of type casting is necessary in C++.

In C++,

```

#include <iostream>
using namespace std;
int main()
{
    void *ptr; // void pointer declaration
    int *ptr1; // integer pointer declaration
    int data=10; // integer variable initialization
    ptr=&data; // storing the address of data variable in void pointer variable
    ptr1=(int *)ptr; // assigning void pointer to integer pointer
    std::cout << "The value of *ptr1 is : " <<*ptr1<< std::endl;
    return 0;
}

```

Output

```
▼ ↗ 📄  
The value of *ptr1 is : 10  
  
...Program finished with exit code 0  
Press ENTER to exit console.□
```

POINTER VERSUS REFERENCE

Pointer holds the memory address of a variable.	Reference is an alias for another variable.
An indirection operator * is used to dereference a pointer.	Reference is a constant pointer which doesn't need a dereferencing operator.
It's an independent variable which can be reassigned to refer to different objects.	It must be assigned at initialization and once created, the address values cannot be reassigned.
NULL value can be assigned to a pointer variable directly.	NULL value cannot be assigned directly.
It lacks automatic indirection.	Automatic indirection is convenient.

DifferenceBetween.net

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int a=10;  
    int &value=a;  
    std::cout << value << std::endl;  
    return 0;  
}
```

Output

```
10
```

```

#include <iostream>
using namespace std;
int main()
{
    int a=9; // variable initialization
    int b=10; // variable initialization
    swap(a, b); // function calling
    std::cout << "value of a is :" <<a<< std::endl;
    std::cout << "value of b is :" <<b<< std::endl;
    return 0;
}

void swap(int &p, int &q) // function definition
{
    int temp; // variable declaration
    temp=p;
    p=q;
    q=temp;
}

```

Output

```

value of a is :10
value of b is :9

```

- In the case of References, reference to reference is not possible. If we try to do c++ program will throw a compile-time error

Pointer Arithmetic Operations

```
#include <iostream>
using namespace std;
int main()
{
    int a[]={1,2,3,4,5}; // array initialization
    int *ptr; // pointer declaration
    ptr=a; assigning base address to pointer ptr.
    cout<<"The value of *ptr is : "<<*ptr;
    ptr=ptr+1; // incrementing the value of ptr by 1.
    std::cout << "\nThe value of *ptr is: " <<*ptr<< std::endl;
    return 0;
}
```

Output:

```
The value of *ptr is :1
The value of *ptr is: 2
```

- In references we are not having any reference arithmetic operations.

Address of a function

We can get the address of a function very easily. We just need to mention the name of the function, we do not need to call the function.

Let's illustrate through an example.

```
#include <iostream>
using namespace std;
int main()
{
    std::cout << "Address of a main() function is : " <<&main<< std::endl;
    return 0;
}
```

Calling a function indirectly

```

#include <iostream>
using namespace std;
int add(int a , int b)
{
    return a+b;
}
int main()
{
    int (*funcptr)(int,int); // function pointer declaration
    funcptr=add; // funcptr is pointing to the add function
    int sum=funcptr(5,5);
    std::cout << "value of sum is :." <<sum<< std::endl;
    return 0;
}

```

Output:

```

value of sum is :10

...Program finished with exit code 0
Press ENTER to exit console.

```

Passing a function pointer as a parameter

```

#include <iostream>
using namespace std;
void func1()
{
    cout<<"func1 is called";
}
void func2(void (*funcptr)())
{
    funcptr();
}
int main()
{
    func2(func1);
    return 0;
}

```

What is Memory Management?

Memory management is a process of managing computer memory, assigning the memory space to the programs to improve overall system performance.

In C language, we use the malloc() or calloc() functions to allocate the memory dynamically at run time, and free() function is used to deallocate the dynamically allocated memory. C++ also supports these functions, but C++ also defines unary operators such as new and delete to perform the same tasks, i.e., allocating and freeing the memory. New operator creates an object and delete operator deletes an object. The object that we created using new operator removes only when we use delete operator or else it will be for life time of the program.

New operator

```
int *p = new int;  
float *q = new float;
```

```
int *p = new int(45);  
float *p = new float(9.8);
```

create a single dimensional array

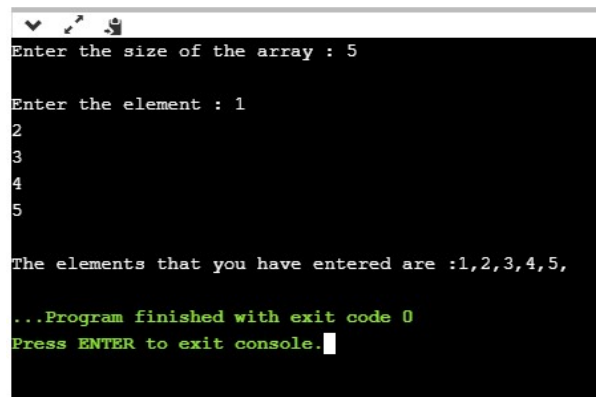
```
int *a1 = new int[8];
```

Delete operator

```
delete p;  
delete q;
```

```
#include <iostream>  
using namespace std  
int main()  
{  
    int size; // variable declaration  
    int *arr = new int[size]; // creating an array  
    cout<<"Enter the size of the array : ";  
    std::cin >> size; //  
    cout<<"\nEnter the element : ";  
    for(int i=0;i<size;i++) // for loop  
    {  
        cin>>arr[i];  
    }  
    cout<<"\nThe elements that you have entered are :";  
    for(int i=0;i<size;i++) // for loop  
    {  
        cout<<arr[i]<<" ";  
    }  
    delete arr; // deleting an existing array.  
    return 0;  
}
```

Output



```
Enter the size of the array : 5

Enter the element : 1
2
3
4
5

The elements that you have entered are :1,2,3,4,5,

...Program finished with exit code 0
Press ENTER to exit console.
```

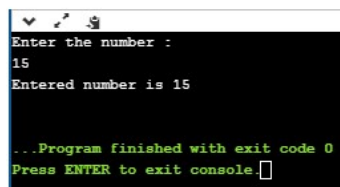
The new is a memory allocation operator, which is used to allocate the memory at the runtime. The memory initialized by the new operator is allocated in a heap. It returns the starting address of the memory, which gets assigned to the variable.

The new operator does not use the sizeof() operator to allocate the memory. It also does not use the resize as the new operator allocates sufficient memory for an object. It is a construct that calls the constructor at the time of declaration to initialize an object.

As we know that the new operator allocates the memory in a heap; if the memory is not available in a heap and the new operator tries to allocate the memory, then the exception is thrown. If our code is not able to handle the exception, then the program will be terminated abnormally.

```
#include <iostream>
using namespace std;
int main()
{
    int *ptr; // integer pointer variable declaration
    ptr=new int; // allocating memory to the pointer variable ptr.
    std::cout << "Enter the number : " << std::endl;
    std::cin >> *ptr;
    std::cout << "Entered number is " << *ptr << std::endl;
    return 0;
}
```

Output:



```
Enter the number :
15
Entered number is 15

...Program finished with exit code 0
Press ENTER to exit console.
```

```

#include <iostream>
#include<stdlib.h>
using namespace std;

int main()
{

    int len; // variable declaration
    std::cout << "Enter the count of numbers : " << std::endl;
    std::cin >> len;
    int *ptr; // pointer variable declaration
    ptr=(int*) malloc(sizeof(int)*len); // allocating memory to the pointer variable
    for(int i=0;i<len;i++)
    {
        std::cout << "Enter a number : " << std::endl;
        std::cin >> *(ptr+i);
    }
    std::cout << "Entered elements are : " << std::endl;
    for(int i=0;i<len;i++)
    {
        std::cout << *(ptr+i) << std::endl;
    }
    free(ptr);
    return 0;
}

```

We can do realloc operation using following process

```

newbuf = new Type[newsize];
std::copy_n(oldbuf, std::min(oldsized, newsize), newbuf);
delete[] oldbuf;
return newbuf;

```

But this method is not suggestable, we can use deque because deque stores its contents non contiguously.

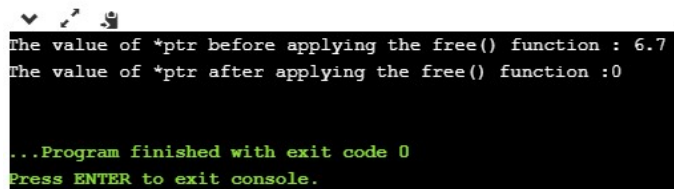
Memory allocated using new operator cannot be relocated.

If the memory is freed which is allocated by malloc then if we try to access that freed memory we would get some garbage value. Whereas if we try to free memory which is allocated by calloc and try to access the freed memory we would get 0 which is default value.


```

#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    float *ptr; // float pointer declaration
    ptr=(float*)calloc(1,sizeof(float));
    *ptr=6.7;
    std::cout << "The value of *ptr before applying the free() function : " <<*ptr<< std::endl;
    free(ptr);
    std::cout << "The value of *ptr after applying the free() function : " <<*ptr<< std::endl;
    return 0;
}

```



```

The value of *ptr before applying the free() function : 6.7
The value of *ptr after applying the free() function : 0

...Program finished with exit code 0
Press ENTER to exit console.

```

Some important points related to delete operator are:

- It is either used to delete the array or non-array objects which are allocated by using the new keyword.
- To delete the array or non-array object, we use delete[] and delete operator, respectively.
- The new keyword allocated the memory in a heap; therefore, we can say that the delete operator always de-allocates the memory from the heap
- It does not destroy the pointer, but the value or the memory block, which is pointed by the pointer is destroyed.

Differences between delete and free()

The following are the differences between delete and free() in C++ are:

- The delete is an operator that de-allocates the memory dynamically while the free() is a function that destroys the memory at the runtime.
- The delete operator is used to delete the pointer, which is either allocated using new operator or a NULL pointer, whereas the free() function is used to delete the pointer that is either allocated using malloc(), calloc() or realloc() function or NULL pointer.
- When the delete operator destroys the allocated memory, then it calls the destructor of the class in C++, whereas the free() function does not call the destructor; it only frees the memory from the heap.
- The delete() operator is faster than the free() function.