1. **Internal identifier :**Identifiers which are used as a local variable or are not used in external linkage are called internal identifiers.

**Eg:** in fn a local variable

1. **External identifier:**Identifiers which are used as a global variable or used for naming function or any other external linkage are called external identifiers.

**Eg:** a variable which is declared as external

While : entry controlled loop, pre test loop

Do-while : exit controlled loop, post-test loop

<https://www.geeksforgeeks.org/functions-in-cpp/> == functions lafda

<https://www.scaler.com/topics/static-member-function-in-cpp/> - static fnctns

// C++ Program to demonstrate working of function using pointers

#include <iostream>

using namespace std;

void fun(int\* ptr) { \*ptr = 30; }

int main()

{

int x = 20;

fun(&x);

cout << "x = " << x;

return 0;

}

// c++ pgm

#include <iostream>

using namespace std;

void fun(int& ptr) { ptr = 30; }

int main(()

{

int x = 20;

fun(x);

cout << "x = " << x;

return 0;

}

Static data member can be called with object as well as without creating object.

In C++, a static member function of a class cannot be virtual. Virtual functions are invoked when you have a pointer or reference to an instance of a class. Static functions aren’t tied to the instance of a class but they are tied to the class. C++ doesn’t have pointers-to-class, so there is no scenario in which you could invoke a static function virtually.

<https://www.geeksforgeeks.org/static-keyword-cpp/>

Diff between virtal fn and pure virtual fn

Abstract class and interface

A class which contains atleast one pure virtual function, is known as abstract class. see the following example.

1)We can’t create an object of abstract class.

1. #include <iostream>
2. **using** **namespace** std;
4. **class** Test {
5. **int** x;
7. **public**:
8. **virtual** **void** show() = 0;
9. **int** getX() { **return** x; }
10. };
12. **int** main(**void**)
13. {
14. Test t;
15. **return** 0;
16. }

2) We can have pointers and references of abstract class type.  
For example the following program works fine.

#include <iostream>

**using** **namespace** std;

**class** Base {

**public**:

**virtual** **void** show() = 0;

};

**class** Derived : **public** Base {

**public**:

**void** show() { cout << "In Derived \n"; }

};

**int** main(**void**)

{

    Base\* bp = **new** Derived();

    bp->show();

**return** 0;

}

**What happens when a virtual function is called inside a non-virtual function in C++**

So polymorphic behaviour works even when a virtual function is called inside a non-virtual function.]

A non-member function can be called inside a member function but the condition is that the non-member function must be declared before the member function.

#include <iostream>

using namespace std;

class sample {

public:

virtual void example() = 0;

};

class Ex1 : public sample {

public:

void example()

{

cout << "GeeksForGeeks";

}

};

class Ex2 : public sample {

public:

void example()

{

cout << " is awesome";

}

};

int main()

{

sample\* arra[2];

Ex1 e1;

Ex2 e2;

arra[0] = &e1;

arra[1] = &e2;

arra[0]->example();

arra[1]->example();

}

**Rules for Virtual Functions**

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have [virtual destructor](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.

Can we make a class constructor *virtual* in C++ to create polymorphic objects? No. C++ being a statically typed (the purpose of RTTI is different) language, it is meaningless to the C++ compiler to create an object polymorphically. The compiler must be aware of the class type to create the object. In other words, what type of object to be created is a compile-time decision from the C++ compiler perspective. If we make a constructor virtual, the compiler flags an error.

Calling virtual methods in constructor/destructor in C++

<https://www.geeksforgeeks.org/calling-virtual-methods-in-constructordestructor-in-cpp/?ref=rp>

Although, bark method is virtual method but when it is called **inside constructor it will behave as non-virtual method because by the time constructor of dog(base) class** is called as in above code, Yellowdog(derived) class is not constructed by that time.

Therefore, it is dangerous to call the member function of class whose object is not constructed yet and compiler calls the dog class version of bark method. And same is with the destructor, when object ‘d’ of Yellowdog gets destroyed, destructor of Yellowdog class is called first and then destructor for dog class is called but by this time Yellowdog is already destroyed, hence dog class version of bark is called.

 It is highly recommended to avoid calling virtual methods from constructor/destructor.

virtual base classes

<https://www.geeksforgeeks.org/virtual-base-class-in-c/?ref=rp>

# Difference Between Friend Function and Virtual Function in C++

<https://www.geeksforgeeks.org/difference-between-friend-function-and-virtual-function-in-cpp/>

# How to access private/protected method of derived class in C++(virtual in base)

# Through virtual function:

<https://www.geeksforgeeks.org/how-to-access-private-protected-method-outside-a-class-in-c/>

Size of Empty Struct in C++ Programming = 1

Size of Empty Struct in C Programming = 0

If observed carefully, the same code is executed in C and C++, but the output is different in both cases. Let’s discuss the reason behind this-

1. The C++ standard does not permit [objects (or classes)](https://www.geeksforgeeks.org/c-classes-and-objects/) of size 0. This is because that would make it possible for two distinct objects to have the same memory location. This is the reason behind the concept that even an empty class and structure must have a size of at least 1. It is known that the size of an empty class is not zero. Generally, it is 1 byte. The C++ Structures also follow the same principle as the C++ Classes follow, i.e. that structures in c++ will also not be of zero bytes. The minimum size must be one byte.

### **Why actually an empty class in C++ takes one byte?**

Simply a class without an object requires no space allocated to it. The space is allocated when the class is instantiated, so 1 byte is allocated by the compiler to an object of an empty class for its unique address identification.

If a class has multiple objects they can have different unique memory locations. Suppose, if a class does not have any size, what would be stored on the memory location? That’s the reason when we create an object of an empty class in a C++ program, it needs some memory to get stored, and the minimum amount of memory that can be reserved is 1 byte. Hence, if we create multiple objects of an empty class, every object will have a unique address.

#include <iostream>

**using** **namespace** std;

// Creating an Empty Class

**class** Empty\_class {

};

// Driver Code

**int** main()

{

    cout << "Size of Empty Class is = "

         << **sizeof**(Empty\_class);

**return** 0;

}

# Publicly inherit a base class but making some of public method as private

<https://www.geeksforgeeks.org/publicly-inherit-base-class-making-public-method-private/>

define the public methids of base class as

 // derived class.

**using** Base::showBaseProperties;

**using** Base::setBaseProperties;

in private section of derived class

# Functions that cannot be overloaded in C++

<https://www.geeksforgeeks.org/function-overloading-in-c/>

**static** **void** fun(**int** i) {}

**void** fun(**int** i) {}

**int** foo() {

**return** 10;

}

**char** foo() {

**return** 'a';

}

**int** f ( **int** x, **int** y) {

**return** x+10;

}

**int** f ( **int** x, **int** y = 10) {

**return** x+y;

}

**int** f ( **int** x) {

**return** x+10;

}

**int** f ( **const** **int** x) {

**return** x+10;

}

# Can main() be overloaded in C++?

<https://www.geeksforgeeks.org/can-main-overloaded-c/>

In C++, what is the difference between exit(0) and return 0 ?

When exit(0) is used to exit from program, destructors for locally scoped non-static objects are not called. But destructors are called if return 0 is used.

# exit(0) vs exit(1) in C/C++ with Examples

[exit](https://www.geeksforgeeks.org/exit-vs-_exit-c-cpp/) is a jump statement in C/C++ language which takes an integer (zero or non zero) to represent different exit status.

There are two types of exit status in C/C++:

1. **Exit Success:** **Exit Success** is indicated by **exit(0)** statement which means successful termination of the program, i.e. program has been executed without any error or interrupt.
2. **Exit Failure:** **Exit Failure** is indicated by **exit(1)** which means the abnormal termination of the program, i.e. some error or interrupt has occurred. We can use different integer other than 1 to indicate different types of errors.

<https://www.geeksforgeeks.org/exit0-vs-exit1-in-c-c-with-examples/>

# return 0 vs return 1 in C++

<https://www.geeksforgeeks.org/return-0-vs-return-1-in-c/>

# exit() vs \_Exit() in C/C++

<https://www.geeksforgeeks.org/exit-vs-_exit-c-cpp/>

# Difference between exit() and break in C/C++

<https://www.geeksforgeeks.org/difference-between-exit-and-break-in-c-cpp/>

# return statement vs exit() in main()

<https://www.geeksforgeeks.org/return-statement-vs-exit-in-main/>

# How can we write main as a class in C++?

<https://www.geeksforgeeks.org/how-can-we-write-main-as-a-class-in-c/>

# How to Access Global Variable if there is a Local Variable with Same Name in C/ C++?

Extern and ::

Why do we create pointer object of an interface :

When you create a pointer object from an interface in C++, you are essentially creating a pointer to an object of a class that implements the interface. This pointer can be used to access the methods and properties of the object through the interface.

In C++, creating pointer objects of a class allows for more flexibility and control over how the class is used and manipulated in memory. Here are a few reasons why you might want to use pointer objects of a class in C++:

1. Dynamic memory allocation: When you create a pointer object of a class, you can allocate memory for the object dynamically at runtime using the `new` operator. This allows you to create objects of varying sizes, depending on the needs of your program.

2. Object lifetime management: By creating a pointer object of a class, you have more control over when the object is created and destroyed. You can use the `new` operator to create the object at a specific point in your program, and then use the `delete` operator to destroy the object when it is no longer needed. This can help to prevent memory leaks and ensure that your program runs efficiently.

3. Object sharing and passing by reference: When you pass an object of a class to a function, a copy of the object is created. However, when you pass a pointer object of a class to a function, you are passing a reference to the original object in memory. This allows multiple functions to share and manipulate the same object, which can help to reduce memory usage and improve performance.

4. Polymorphism: When you create a pointer object of a class, you can use polymorphism to treat objects of different derived classes as if they were of the same type. This allows you to write more generic and reusable code that can work with different types of objects.

Overall, creating pointer objects of a class in C++ provides greater flexibility and control over how the class is used and manipulated in memory, and can help to improve the efficiency and performance of your program.

In C++, passing a structure as a `const` parameter in a function has several advantages:

1. Preventing modification: By declaring a structure parameter as `const`, you indicate to the compiler that the function will not modify the contents of the structure. This can help prevent inadvertent modification of the structure, which can be particularly useful when passing structures by reference or pointer.

2. Improving performance: When a structure is passed by value, a copy of the structure is created. However, when a structure is passed as a `const` reference, only a reference to the original structure is passed, which can be more efficient in terms of memory usage and performance.

3. Enforcing code clarity: Declaring a structure parameter as `const` can also help to enforce clarity in your code by indicating to other programmers that the structure will not be modified by the function. This can make it easier for others to understand your code and help to prevent unintended modifications of the structure.

4. Compatibility with other code: Some libraries and functions may require `const` parameters for compatibility reasons. By using `const` parameters for your structures, you can ensure that your code is compatible with these libraries and functions.

In general, passing structures as `const` parameters in C++ can help improve code clarity, prevent unintended modifications, and improve performance.

Arrays :

* In C/C++, initialization of a multidimensional arrays can have left most dimension as optional. Except the left most dimension, all other dimensions must be specified.

Eg: int arr[[[3] { }; /////////////////////////////////////////////////////////////////

* int a[3] {};
* cout<<a[10]; //garbage value not error

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

------- >>>> Finding the size of an array using fn’s

#include <iostream>

**using** **namespace** std;

**void** findSize(**int** arr[])

{

    cout << **sizeof**(arr) << endl;

}

**int** main()

{

**int** a[10];

    cout << **sizeof**(a) << " "; // 40 8

    findSize(a);

**return** 0;

}

The above output is for a machine where the size of an integer is 4 bytes and the size of a pointer is 8 bytes.  
The **cout** statement inside main prints 40, and **cout** in findSize prints 8. The reason is, arrays are always passed pointers in functions, i.e., findSize(int arr[]) and findSize(int \*arr) mean exactly same thing. Therefore the cout statement inside findSize() prints the size of a pointer.

Therefore we pass array to function as pass by reference:

**void** findSize(**int** (&arr)[10]);

getline()

<https://www.geeksforgeeks.org/write-a-c-program-that-wont-compile-in-cpp/>

<https://www.geeksforgeeks.org/extern-c-in-c/>

exit(0) – indicate the succeful execution of program2

exit(1) – indicates the abnormal termination of a program

When exit(0) is used to exit from program, destructors for locally scoped non-static objects are not called. But destructors are called if return 0 is used.

**typedef** keyword is used to assign a new name to any existing data-type.

For example, if we want to declare some variables of type **unsigned int**, we have to write *unsigned int* a, we can use : : typedef unsigned int unit;

uint a;

Returning multiple values from functions:

1. Passing arguments as reference
2. Through arryas
3. Through structure
4. Using class and objects

<https://www.geeksforgeeks.org/how-to-return-multiple-values-from-a-function-in-c-or-cpp/?ref=rp>

Exception Handling:

<https://www.geeksforgeeks.org/exception-handling-c/>

<https://www.geeksforgeeks.org/c-plus-plus-gq/exception-handling-gq/>

**9)**When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block

#include <iostream>

using namespace std;

class Test {

public:

Test() { cout << "Constructor of Test " << endl; }

~Test() { cout << "Destructor of Test " << endl; }

};

int main()

{

try {

Test t1;

throw 10;

}

catch (int i) {

cout << "Caught " << i << endl;

}

}

o/p

Constructor of Test

Destructor of Test

Caught 10