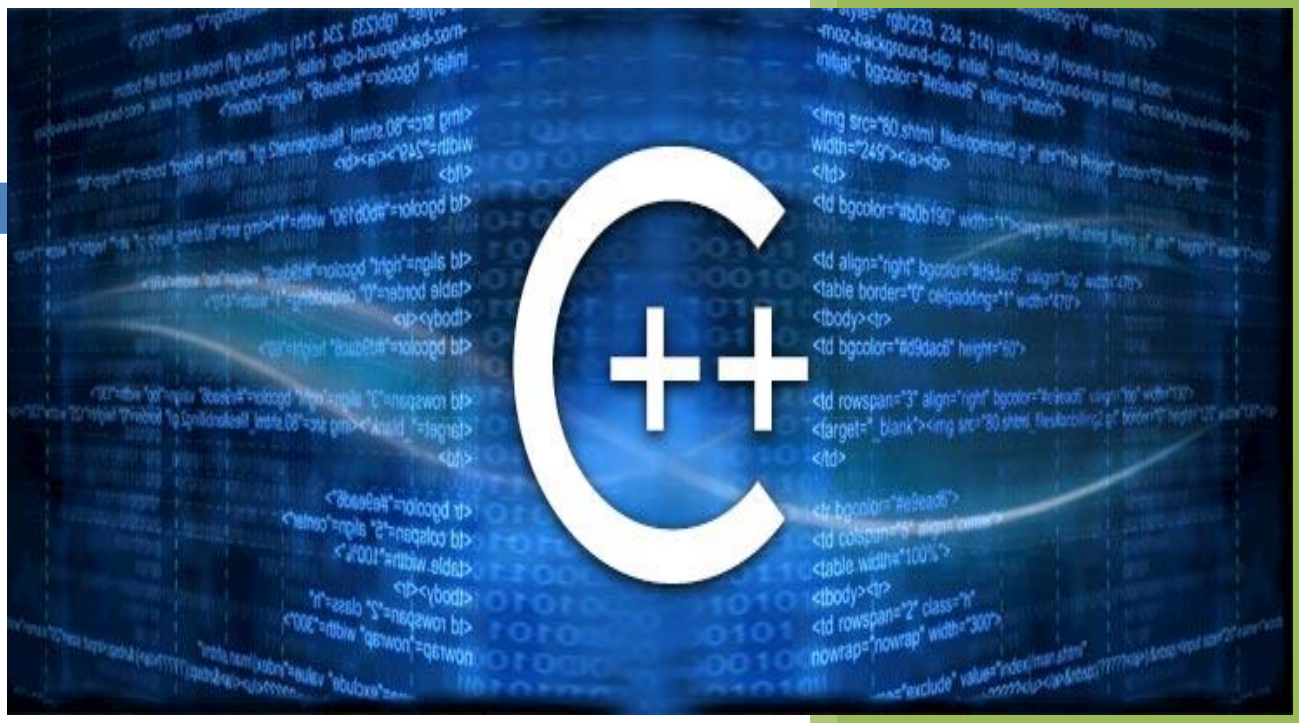
2016



Contents

Basics

• About C++

- Developed by **Bjarne stroustrup** of **AT & T Bell** laboratory in 1980.
- It has imperative, object-orientedand generic programming features, while also providing facilities for low-level memory manipulation.
- C++ is derived from C Language. It is a Superset of C.
- All C operators are valid in C++.

❖ Difference with C

C	C++
C is Procedural Language.	C++ is non Procedural i.e Object oriented language.
Prototype optional.	Prototype compulsory
No virtual Functions are present in C.	The concept of virtual Functions are used in C++.
Top down approach is used in Program Design.	Bottom up approach adopted in Program Design.
In C, we can call main() Function through other Functions	In C++, we cannot call main() Function through other functions. Callable from main function.
It supports built-in and primitive data types.	It support both built-in and user definedata types.
In C, Exception Handling is not present.	In C++, Exception Handling is done with Try and Catch block.

❖ Features & key-points

- **Simple**: It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich library support and a variety of data-types.
- **Machine Independent but Platform Dependent**: A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.
- **Mid-level language**: It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)
- **Rich library support**: Has a rich library support (Both standard ~ built-in data structures, algorithms etc.) as well 3rd party libraries (e.g. Boost libraries) for fast and rapid development.
- **Speed of execution**: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as garbage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.
- **Pointer and direct Memory-Access**: C++ provides pointer support which aids users to directly manipulate storage address. This helps in doing low-level programming (where one might need to have explicit control on the storage of variables).

- **Object-Oriented:** One of the strongest points of the language which sets it apart from C. Object-Oriented support helps C++ to make maintainable and extensible programs. i.e. Large-scale applications can be built. Procedural code becomes difficult to maintain as code-size grows.

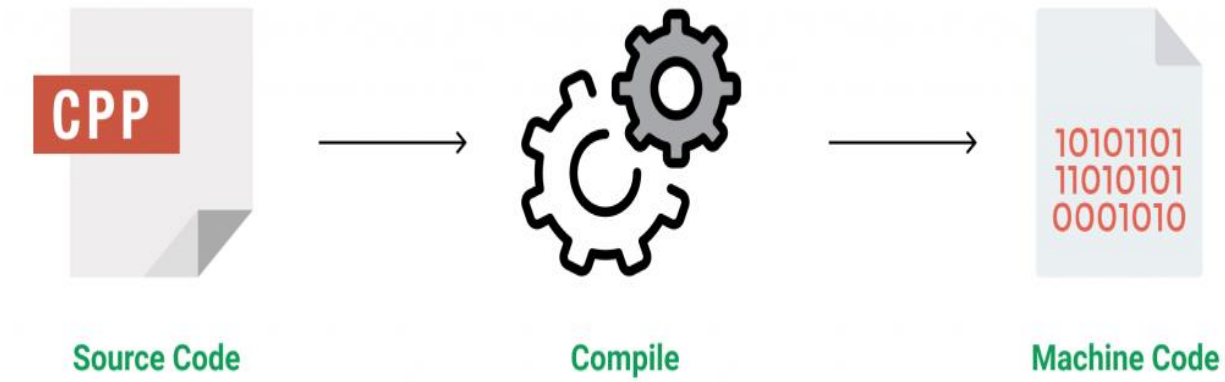
❖ **Compiled Language**: C++ is a compiled language, contributing to its speed.

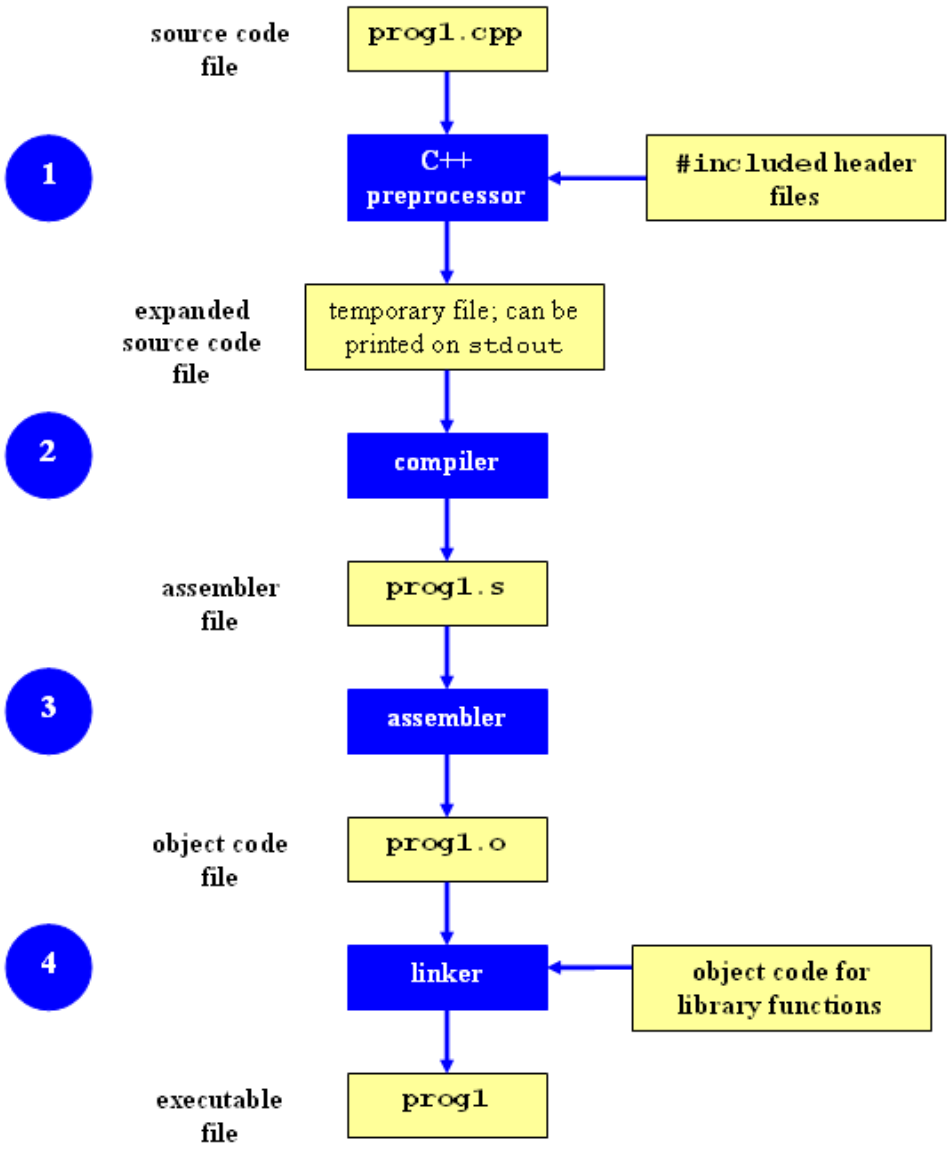
❖ First C++ program

```
// my first program in C++
#include <iostream>
int main ()
{
  std::cout <<"Hello World! ";
  std::cout <<"I'm a C++ program";
}
```

❖ The C++ compilation & linking process

- **Preprocessing**: the preprocessor takes a C++ source code file and deals with the #includes, #defines and other preprocessor directives. The output of this step is a "pure" C++ file without pre-processor directives.
- **Compilation**: the compiler takes the pre-processor's output and produces an object file from it.
- **Assembler** : The assembler code generated by the compiler is assembled into the object code for the platform.
- **Linking**: the linker takes the object files produced by the compiler and produces either a library or an executable file. It resolves external references & make sure that external functions & data existed. It also adds a special object module to perform start up activities.





❖ Why Java is Platform Independent and C, C++ not !!

Platform Independent Vs Platform Dependent

Platform Independent means the program that we have developed can run / execute (show results) on any platform. That is on any operating system.
Platform Dependent means the program / software that we have developed can run /execute (show results) on a specific platform. That is on a specific operating system.

Why Java is Platform Independent and C, C++ not !!

The programming language like **C and C++ do not generate any intermediate code like bytecode in java**. The C and C++ generate directly Native Code which is specific to one platform or operating system. This Native code is known as **object code (Machine Code).**

As we also know that Object Code (Machine Code) **generation requires some Operating system files to be included which makes it platform dependent.**

Whereas in Java we have intermediate code by the name of **Byte Code** (not a machine code). This can is understandable by **JVM** (Java virtual Machine) **which converts it to the object code** (machine Code) . That why in order to run **Java code we need JVM installed** on the device which makes java Platform independent.

When you compile and link a C++ program it create an executable image containing code which could be natively run on the processor, whereas when you compile a java file it generates class file containing Java Byte Code which requires the JVM to convert it to machine readable format.

If you are writing a program purely using C++, you can recompile it without any changes on any platform and then run it there. The problem will arise if you are using any **OS specific libraries or code** inside your C++ code, in such cases you **can't directly compile** on other platforms.

As you can see applications written using C++/C will have a advantage over Java in term of performance (no extra **overhead of JVM**), also you **won't be able to write system level apps using Java**. There are various other pros and cons for using Java or C++, and this is generally dictated by the type of application you are trying to develop.

A C++ program may or may not be platform-dependent. **If you call functions that are provided by the compiler on Windows and not in other operating systems,** then your C++ program will compile only on Windows. That is a platform dependence. But if you avoid calling platform-dependent functions such as those, you can compile the C++ program anywhere.

❖ Including header in C++

Usual practice is,

- Use angle brackets for system headers. E.g. #include <stdlib.h>
- User double quotes for User defined headers(Your own headers). e.g : #include "Some_Header.h"

❖ Some points about C++

- cin- Console input
- cout- Console output
- Both cin & cout desined in #include<iostream> file

• Bitwise Operators in C/C++

- The & (**bitwise AND**) in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

- The | (**bitwise OR**) in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

- The ^ (**bitwise XOR**) in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

- The << (**left shift**) in C or C++ takes two numbers, left shifts the bits of the first operand, the second

operand decides the number of places to shift.

- The >> (**right shift**) in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- The ~ (**bitwise NOT**) in C or C++ takes one number and inverts all bits of it

❖ Interesting facts about bitwise operators

- **The left shift and right shift operators should not be used for negative numbers.**

For example results of both 1 <<- 1 and 1 >> -1 is undefined. Also, if the number is shifted more than the size of the integer, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits.

- **The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.**

```
 int x = 19;
 cout << "x << 1 = " << (x << 1) << endl; // O/P: 38
 cout << "x >> 1 = " << (x >> 1) << endl; // O/P: 9
```

- **The & operator can be used to quickly check if a number is odd or even.**

```
int x = 19;
(x & 1) ? cout << "Odd" : cout << "Even"; // O/P: Odd
```

- **We can use XOR to if two numbers are equal without using the equal to operator**

```
void areSame(int a, int b)
{
    if (a ^ b) // The result of XOR is 1 if the two bits are different.
        cout << "Not Same";
    else
        cout << "Same";
}
```

- **We can swap the two nubers with XOR like this**

```
void swap(int a, int b)
{
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

- **We can print kth bit of number like this…**

```
void printKthBit(unsigned int n, unsigned int k)
{
    cout << ((n & (1 << (k - 1))) >> (k - 1));
}
```

- **Function to set the kth bit**

```
int setKthBit(int n, int k)
{
    // kth bit of n is being set by this operation
    return ((1 << k-1) | n);
```

```
}
```

- **Function to clear the kth bit of n**

```
int clearBit(int n, int k)
{
    return (n & (~(1 << (k - 1))));
}
```

- **Function to toggle the kth bit of n**

```
int toggleBit(int n, int k)
{
    return (n ^ (1 << (k - 1)));
}
```

Object Oriented Programming

Advantages:

- Data is more important
- Map with real world.
- Reusing the code
- Complexity reduced

❖ Class

- A class is a design or specification of particular category.
- Class is user defined data type.

❖ Object

- Instance of a class.

C++ struct	C++ class
Default access is public.	Default access is private.
Used to store data.	Used to store data & operations on the data.

C struct	C++ struct
Only data is stored.	Both data & function can be stored.
Keyword struct is compulsory. e.g. struct student a;	Keyword struct is optional. e.g student a;

❖ Constructor

- Function with same name as of class.
- Used to initialize data.
- Executes automatically.
- Cannot be declared as static , constant or virtual.
- Should have public or protected access.

Constuctor	Destructor
Called when object created.	Called when object goes out of scope.
Job is initialization.	Job is to clean up
Can take parameter.	Cant take parameter.
Can be overloaded	Cant be overloaded

❖ Default Constructor

- If we don't write constructor, C++ internally provides no parameter constructor which initializes all data with 0.

❖ Placement new operator in C++

Placement new is a variation new operator in C++.
Normal new operator does two things :

1. Allocates memory

2. Constructs an object in allocated memory.

Placement new allows us to separate above two things. In placement new, ***we can pass a preallocated memory and construct an object in the passed memory.***

Key Notes:

- Normal new allocates memory in heap and constructs objects there whereas using ***placement new***, object construction can be done at ***known address***.
- With normal new, it is not known that, at what address or memory location it's pointing to, whereas the address or memory location that it's pointing is known while using placement new.
- The deallocation is done using delete operation when allocation is done by new but there is no placement delete, but if it is needed one can write it with the help of destructor.

Syntax:

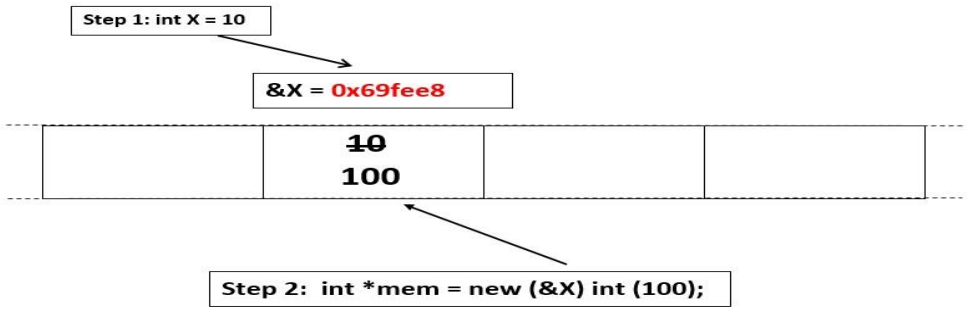
```
new (address) (type) initializer
```

e.g:

```
// C++ program to illustrate the placement new operator
#include<iostream>
using namespace std;
int main()
{
    // initial value of X
    int X = 10;

    cout << "Before placement new :" << endl;
    cout << "X : " << X << endl;
    cout << "&X : " << &X << endl;

    // Placement new changes the value of X to 100
    int* mem = new (&X) int(100);

    cout << "\nAfter placement new :" << endl;
    cout << "X : " << X << endl;
    cout << "mem : " << mem << endl;
    cout << "&X : " << &X << endl;
    // Here we don't need to delete the memory as it is stack memory.
    // If heap memory is used in placement new then we need to delete it.
    return 0;
}
```

O/P:

```
Before placement new :
X : 10
&X : 0x69fee8


After placement new :
X : 100
mem : 0x69fee8
&X : 0x69fee8
```



e.g:2

```
class Base
{
public:
    Base() { cout << constuctor << endl; }
    ~Base() { cout << destructor << endl; }
};

int main()
{
    // Normal case
    Base* obj = new Base();
    delete obj;

    // Placement new case
    char* memory = new char[10 * sizeof(Base)]; //Reserve memory for 10 Base object

    Base* obj1 = new(&memory[0])Base();
    Base* obj2 = new(&memory[4])Base();
    Base* obj3 = new(&memory[8])Base();

    obj1->~Base();
    obj2->~Base();
    obj3->~Base();
    delete[] memory;
    return 0;
}
```

When to prefer using placement new?
As it allows to construct an object on memory that is already allocated, ***it is required for optimizations as it is faster not to re-allocate all the time***. There may be cases when it is required to ***re-construct an object multiple times so, placement new operator might be more efficient*** in these cases.

When will placement new operator show segmentation fault?
The placement new operator should be used with care. The address which is passed can be a reference or a pointer pointing to a valid memory location. It may show an error when the address passed is :

- A pointer such as **NULL pointer**.
- A pointer that is not pointing to any location.

- It cannot be a void pointer unless it points to some location.

Advantages of placement new operator over new operator

- The address of memory allocation is known before hand.
- Useful when **building a memory pool,** a garbage collector or simply when performance and exception safety are paramount.
- There's **no danger of allocation failure** since the memory has already been allocated, and constructing an object on a pre-allocated buffer takes less time.
- This feature becomes useful while working in an environment with limited resources.

❖ Understanding nullptr in C++

- NULL is typically defined as (void *)0 and conversion of NULL to integral types is allowed.

```
// function with integer argument
void fun(int N) { cout << "fun(int)"; return; }

// Overloaded function with char pointer argument
void fun(char* s) { cout << "fun(char *)"; return; }

int main()
{
    // Ideally, it should have called fun(char *),
    // but it causes compiler error.
    fun(NULL);
}
```

O/P:

```
16:13: error: call of overloaded 'fun(NULL)' is ambiguous fun(NULL);
```

What is the problem with above program?

NULL is typically defined as (void *)0 and conversion of NULL to integral types is allowed. So the function call fun(NULL) becomes ambiguous.

How does nullptr solve the problem?

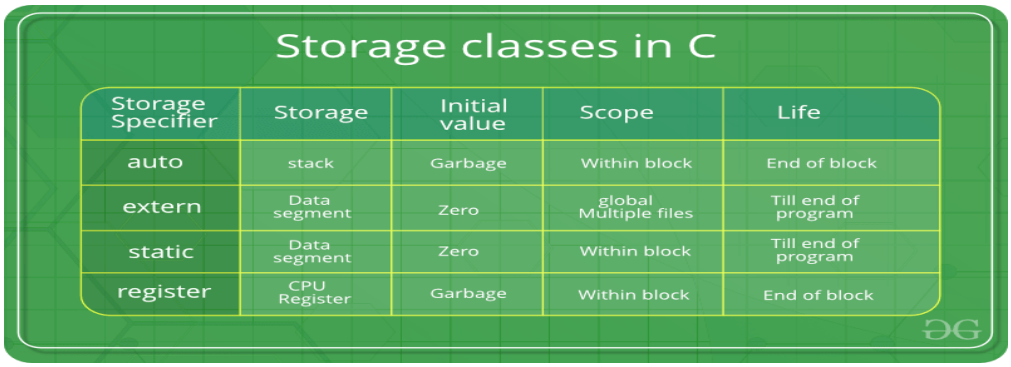
In the above program, if we replace NULL with nullptr, we get the output as **"fun(char *)".**

nullptr is a keyword that can be used at all places where NULL is expected. Like NULL, nullptr is implicitly convertible and comparable to any pointer type.

Unlike NULL, it is not implicitly convertible or comparable to integral types.

```
int x = nullptr; // Error


int* ptr = nullptr; //OK

// Below line compiles
if (ptr) { cout << "true"; } // OK
else { cout << "false"; }
```

❖ Storage Classes



Constructors And Destructor in C++

• Constructor in C++

A constructor is a member function of a class which initializes objects of a class.
A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself.
- Constructors don't have return type.
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

e.g.

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

class Point
{
private:
    int x, y;
public:
    /***Constructor****/
    Point(int x1, int y1) { x = x1; y = y1; }

    int getX()            {  return x; }
    int getY()            {  return y; }
};
```

- Value of data members is undefined if we write our own constructor and does not assign any value & even for default constructor.

```
class Test
{
    int var;
public:
    Test() {}
};

int main()
{
    Test* t1 = new Test; // Val of var = udefined.
    Test* t2 = new Test();// Val of var = 0.
    Test t3; // Val of var = udefined.
}
```

• Destructors in C++

Destructor is a member function which destructs or deletes an object.
A destructor function is called automatically when the object goes out of scope:

- The function ends.
- The program ends.
- A block containing local variables ends.
- A delete operator is called.

E.g.

```
class String
{
private:
    char *s;
    int size;
public:
    String(char *); // constructor
    ~String();      // destructor
};

String::String(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}

String::~String()
{
    delete []s;
}
```

Can there be more than one destructor in a class?
No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type

Can we call destructor on own?

```
class Base
{
public:
    ~Base() { cout << "Dest"; }
};

int main()
{
    Base b;
    b.~Base(); //
}
```

O/P:

```
Dest Dest
```

Two times destructor gets called as b is local variable and when it goes out of scope again one more time destructor gets called. This is dangerous if we are deleting any pointer inside destructor.

• Copy Constructors in C++

A copy constructor is a member function which initializes an object using another object of the same class. If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects.

A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

In C++, a Copy Constructor may be called in following cases:

- When an object of the class is returned by value.
- When an object of the class is passed (to a function) by value as an argument.
- When an object is constructed based on another object of the same class.
- When compiler generates a temporary object.

Suppose class

```
class Obj
{
    int val;
public:
    Obj(int val) //Parameterized Constructor
    {
        this->val = val;
    }

    Obj(const Obj& o) //Copy Constructor
    {
        this->val = o.val;
    }
}

int main(int argc, _TCHAR* argv[])
{
    Obj a(10); //Calls Parameterized Constructor
    Obj b = 20; //Calls Parameterized Constructor this is same as Obj b(20);
    Obj c = a; //Calls Copy Constructor
    Obj d = Obj(10); //Calls Parameterized Constructor this is same as Obj d(10);
    Obj e = 'a';  // Calls Parameterized Constructor this is same as Obj e(97);
    Obj f = "sgy"; // Compiler error No sutaible constructor which takes char array as
parameter.
    e = a; //Will call assignment operator.
}
```

If object is created like Obj p = something then something is passed as parameter to one of the constructor in case of object b 20 is passed to parameterized constroctor & in case of c object a is passed to copy constructor & in case of d value 10 is passed to parameterized constructor. So anything which is other side of = is passed as param to constructor.

❖ **Copy constructor vs Assignment Operator**

```
MyClass t1, t2;
MyClass t3 = t1;  // ----> (1) Copy Constructor
t2 = t1;          // -----> (2) Assignment Operator
```

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. Assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls copy constrictor and (2) calls assignment operator. See this for more details.

❖ Can we make copy constructor private?

Yes, *a copy constructor can be made private*. When we make a copy constructor private in a class, objects of that *class become non-copyable*. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like above String example, or make a private copy constructor so that users get compiler errors rather than surprises at run

time.

❖ Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be pass by value.

❖ Why is the argument of the copy constructor a reference rather than a pointer?

- ***References cannot be NULL.*** References are therefore always expected to refer to actual objects.
- Passing by references ensures an actual object is passed to the copy constructor, whilst a pointer can have NULL value, and make the constructor fail.

```
Obj(const Obj* o)
{
    this->val = o->val;
}
```

This is also looks like copy constructor. Only downside is pointer may come **NULL** so constructor might fail.

```
class A
{
public:
    ~A()
    {
        i = 10;
    }
};

int foo()
{
    i = 3;
    A obj;
    return i;
}
```

O/P of foo is : 3
While returning from a function, destructor is the last method to be executed. The destructor for the object "obj" is called after the value of i is copied to the return value of the function. So, before destructor could change the value of i to 10, the current value of i gets copied & hence the output is i = 3.

If function is like

```
int foo()
{
    i = 3;
    {
        A ob;
    }
    return i;
}
```

O/P of foo is : 10

Since the object ob is created in the block scope, the destructor of the object will be called after the block ends, thereby changing the value of i to 10. Finally 10 will copied to the return value.

Or

```
int& foo()
{
    i=3;
    A obj;
    return i;
}
```

The function foo() returns the l-value of the variable i. So, the address of i will be copied in the return value. Since, the references are automatically dereferened. It will output “i = 10”.

❖ What are all the member-functions created by compiler for a class?

- default constructor
- copy constructor
- copy operator (assignment)
- destructor

Invalid copy constructor

```
class Point {
    int arr;
public:
    Point(int x) {
        arr = x;
    }
    Point(Point val) {}
}
```

```
o/p: Compilation error. Copy constructor is not valid.
```

• Explicit Constructor

- Some times we need explicit constructor in c++ programming language, and we will see what is the need to use explicit keyword at front of constructor in c++. It is used to avoid some inbuilt feature of language, which sometimes create confusion.

```
#include <iostream>
class Base
{
    int b_var;
public:
    Base() {}
    Base(int var) : b_var(var) {}
    void print() { cout << b_var << endl; }
};
void fun(Base b)
{
    b.print()
}

int main()
{
    Base obj1(10); // Normal call to constuctor
    Base obj2 = 10; // Implicit call to constuctor

    fun(ob1); // Normal call to constuctor
    fun(30); // Implicit call to constuctor
```

```
    return 0;
}
O/P:
    10
    30
```

So to **avoid such confusion** we can make constructor **explicit** to the implicit conversion is not possible and compiler will give error for those line.

```
explicit Base(int var) : b_var(var) {}
```

Now

```
Base obj2 = 10;
fun(30);
```

These two lines will give **compilation error.**

We need to call them like,

```
Base obj2 = Base(20);
fun(Base(30));
```

Object Oriented Approach

❖ Encapsulation
- Bind together code & data.
- Keep both safe from misuse.

❖ Polymorphism
- Quality of having more than one form.
- Single operation can have diff. behavior in diff objects.

❖ Inheritance
- One object can acquire properties of another object.

❖ Abstraction
- Ability to represent a complex problem in simple term.

❖ Specialization
- Ability to make lower level sub-classes more detailed.

❖ Command Line Arguments
It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

e.g :
```
#include <stdio.h>
int main( int argc, char *argv[] )  {
   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
}
```

❖ Manipulators
- endl – New line
- setw –Set width
- e.g setw(5)
- flush- flushes buffer.
- Dec – Outputs in decimal
- Hex – Outputs in hex
- Oct– Outputs in oct

❖ Enum
- Allow to define symbolic constants.
  e.g: enum

❖ Converting Strings to Numbers in C/C++

Using sscanf

```
sscanf(str, "%d", &x);
```

Using atoi

atoi() : The atoi() function takes a character array or string literal as an argument and returns its value.

```
int num1 = atoi(str1);
```

Using stoi

The stoi() function takes a string as an argument and returns its value.

```
int myint1 = stoi(str1);
```

stoi() vs atoi()

- atoi() is a legacy C-style function. stoi() is added in C++ 11.
- atoi() works only for C-style strings (character array and string literal), stoi() works for both C++ strings and C style strings

❖ Constant Variable

- Const float pi;
  This gives error because ***const must be declared at time of definition***.
- Value of constant ***cannot be changed***.
- Non const function will not be available to const object whereas const function is available to non-const objects.

1. `const char * p = "Hello"; or`
2. `char * p = "Hello"; // Same as 1 even if you have not mentioned specifically compiler treats this as const string only.`
3. `char const * p = "Hello"; // 1 & 2 & 3 all are same.`
   If const comes before * then string is const & not the pointer.
   (*p cannot be modified)
4. `char * const p ="Hello";`
   Here pointer is const but not string.
   (p cannot be modified like p ="Hi";)
5. `const char * const p = "Hello";`
   Both string & pointers are const.

❖ Const member functions in C++

- A function becomes const when const keyword is used in function's declaration.
- The idea of const functions is not allow them to modify the object on which they are called.

```
// We get compiler error if we add a line like "value = 100;"
   // in this function.
   int getValue() const {return value;}
```

❖ Array

- Stored in continuous memory location.

```
int sample[3] = {5,};  // Will work create sample with 5,0,0

Sample[i] = * (Sample + i );
```

```
Sample[2] = * (Sample + ( 2 * sizeof (int) ) );
Sample[0] = * (Sample + 0 );
```

❖ Constant Object

- Object data members should not be changed.
- Only call to const function using const object.
- A member function that is not specifically declared const is treated as one that will modify data members in an object & compiler will not allow you to call it for const object.

```
class X
{
    int m_data;
public:
    X(int data)
    {
            m_data = data;
    }
    int f1() const
    {
    }
    int f2()
    {
    }
}

int main()
{
    const X x1(10);
    x1.f1(); // Ok
    x1.f2(); //Compiler error.
}
```

❖ Constant Return

- Using this you are promising that the original variable will not be modified & again because you are returning it by value its copied to original value could never be modified via the return value.
- const return values can't be used as l-value.

e.g:

1.

```
const Test func1()
{
        return t1;
}

int _tmain(int argc, _TCHAR* argv[])
{
        Test t2(6);
        func1() = t2; // Not Ok. return type is a const & cant be used as l-
                      value.
                      // It is a temp object crated from t1 with const.
}
```

2.

```
Test func1()
{
        return t1;
```

```
}

int _tmain(int argc, _TCHAR* argv[])
{
	Test t2(6);
	func1() = t2; // Ok. return type is not a const.
		// It is a temp object created from t1 which is non const.
}
```

3.
But for **primitive data type cant be a l-value** so,

```
int func1()
{
	return t1.m_val;
}

int _tmain(int argc, _TCHAR* argv[])
{
	func1() = 6;
// Not Ok func1 return primitive data type which cant be l-value.

}
```

SOLID: The First 5 Principles of Object Oriented Design

SOLID stands for:

- **S** - Single-responsiblity Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

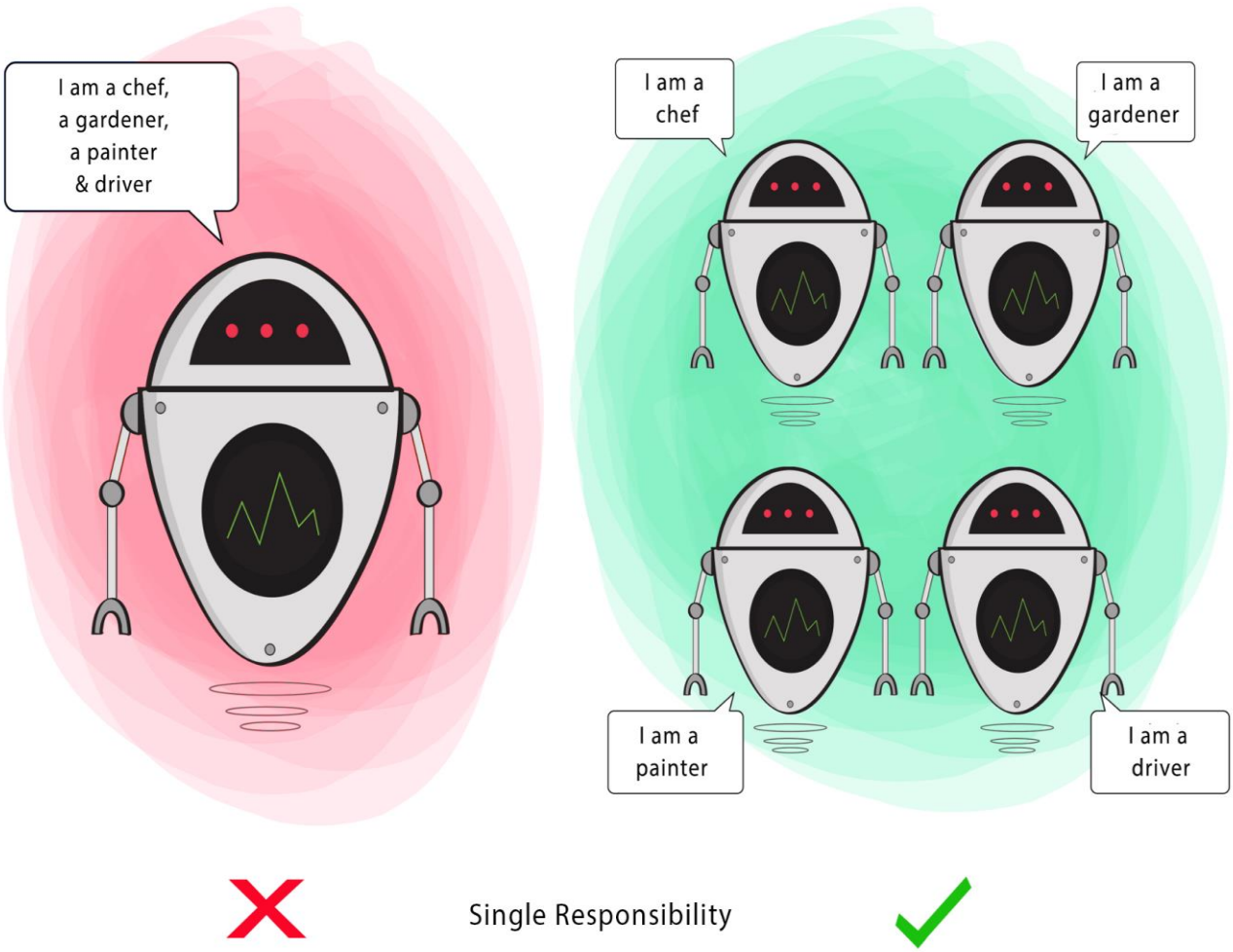
❖ Single-Responsibility Principle

Single-responsibility Principle (SRP) states:

A class should have one and only one reason to change, meaning that a class should have only one job.

E.g: , Consider an application that takes a collection of shapes—circles, and squares—and calculates the sum of the area of all the shapes in the collection.

Next, create the AreaCalculator class and then write up the logic to sum up the areas of all provided shapes. Don't mix shape creation & area calculation in single class. They are two different things.



- **Goal**

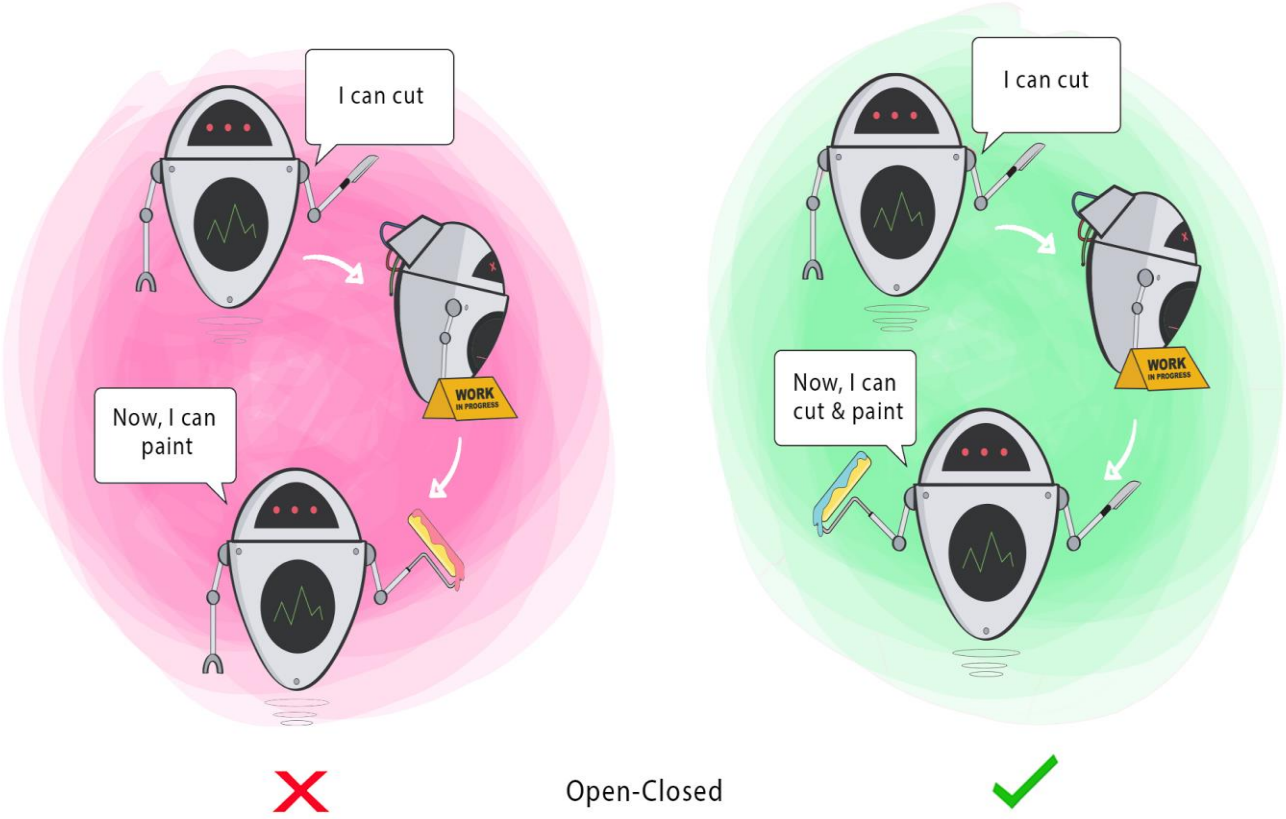
This principle aims to **separate behaviours** so that if bugs arise as a result of your change, it won't affect other unrelated behaviours.

❖ Open-Closed Principle

Open-closed Principle (OCP) states:

Objects or entities should be open for extension but closed for modification.

This means that a class should be **extendable** without modifying the class itself.



- **Goal**

This principle aims to **extend a Class's behaviour without changing the existing behaviour of that Class**. This is to avoid causing bugs wherever the Class is being used.

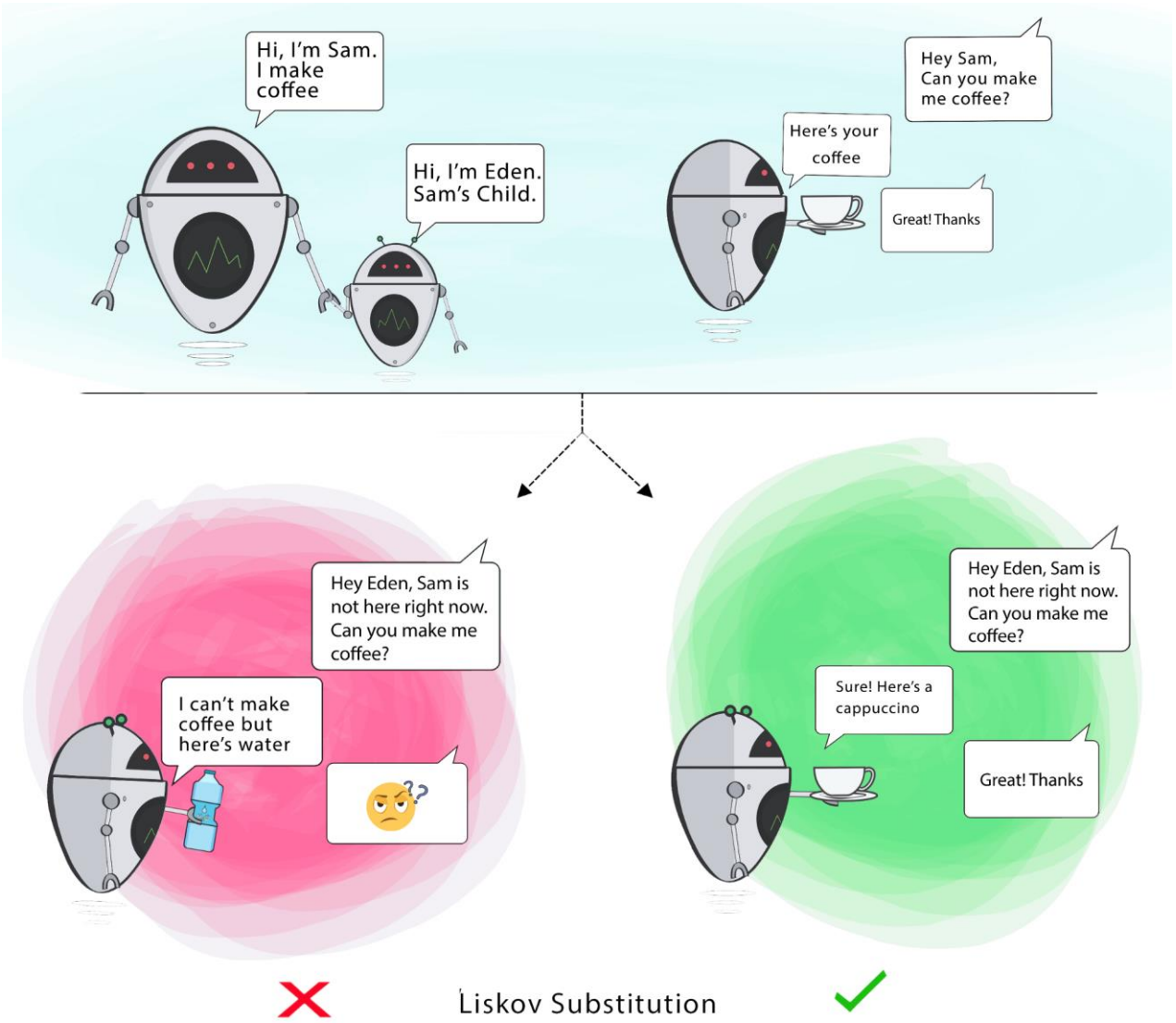
❖ Liskov Substitution Principle

Liskov Substitution Principle states:

Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

In other term,If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

This means that **every subclass or derived class should be substitutable for their base or parent class**.



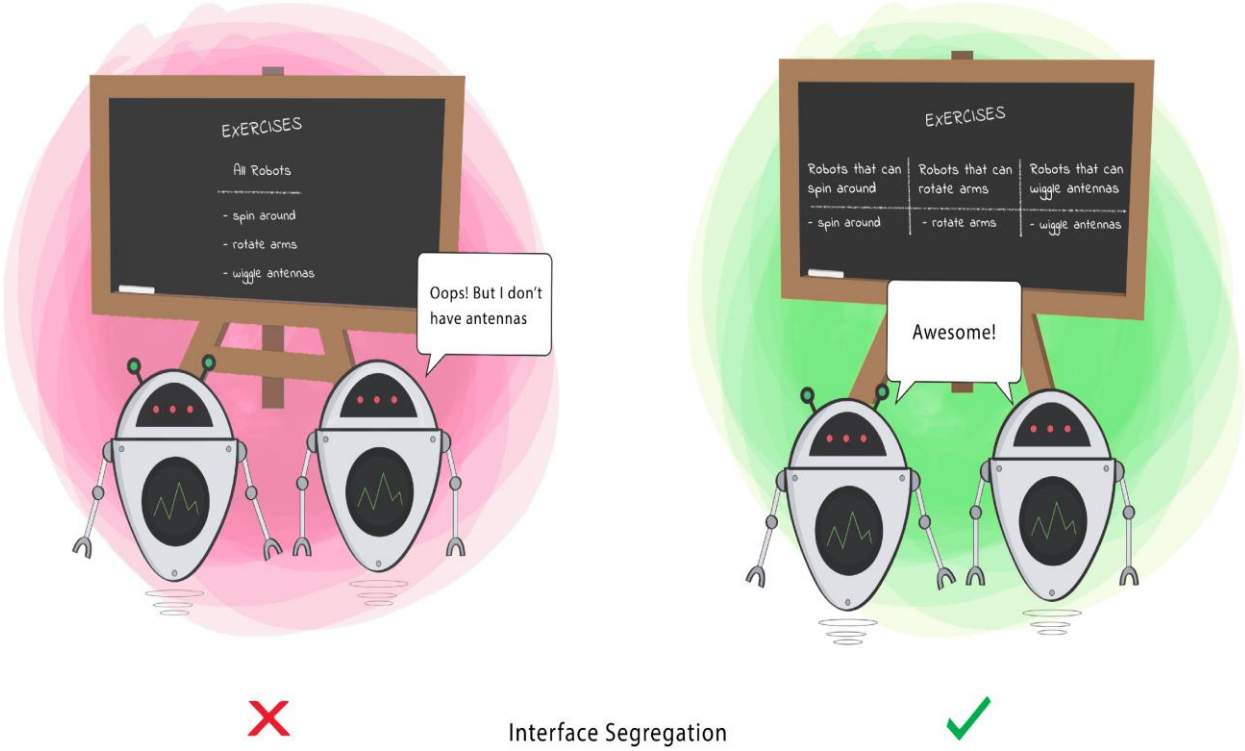
- **Goal**

This principle aims to **enforce consistency so that the parent Class or its child Class can be used in the same way** without any errors.

❖ Interface Segregation Principle

Interface segregation principle states:

A client should **never be forced to implement an interface that it doesn't use**, or clients shouldn't be **forced to depend on methods they do not use**.



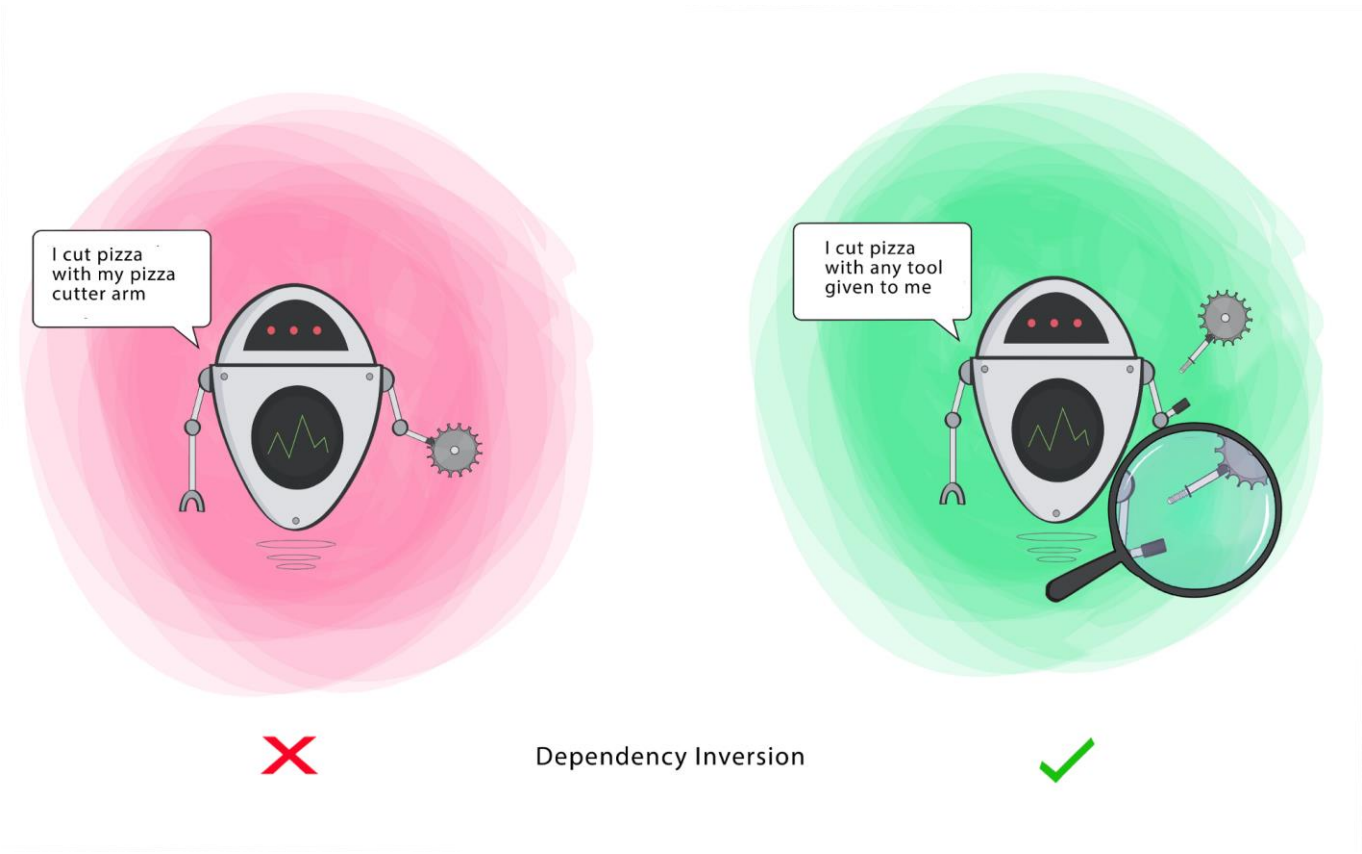
Interface Segregation

- **Goal**

This principle aims at **splitting a set of actions into smaller sets** so that a Class executes ONLY the set of actions it requires.

❖ Dependency Inversion Principle

Dependency inversion principle states:
-Entities must **depend on abstractions, not on concretions**. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.
-High-level modules should not depend on low-level modules. Both should depend on the abstraction.
- Abstractions should not depend on details. Details should depend on abstractions.



- **Goal**

This principle aims at **reducing the dependency of a high-level Class on the low-level Class** by introducing an interface.

Variables

- ## Pointer variable

❖ Pointers basics

A **pointer** is a value that represents the address of an object. Every distinct object has a unique address, which denotes the part of the computer's memory that contains the object.

```
int* p, q; // What does this definition mean?
```

defines p as an object of type "pointer to int" and q as an object of type int. This example is easier to understand if we view it this way:

```
int* p, q; // *p and q have type int
```

or, for that matter, this way:

```
int(*p), q; // (*p) and q have type int
```

We now know enough to write a simple program that uses pointers:

```
int main()
{
    int x = 5;
    // p points to x
    int* p = &x;
    cout << "x = " << x << endl;
    // change the value of x through p
    *p = 6;
    cout << "x = " << x << endl;
    return 0;
}
```

The output of this program will be

```
x = 5
x = 6
```

Immediately after we have defined p, the state of our variables is

Delete nullptr is OK. Does nothing

```
    char* ptr = nullptr;
    delete ptr; // OK
    delete nullptr; // However shows error
```

❖ Pointers to functions

We can call a function with another function as an argument. Pointers to functions behave similarly to any other pointers. Once you have dereferenced such a pointer, however, all you can do with the resulting function is call it

```
int (*fp)(int);
```

to say that if we dereference fp, and call it with an int argument, the result has type int. By implication, fp is a pointer to a function that takes an int argument and returns an int result.

e.g :

```
int next(int n)
{
    return n + 1;
}
```

then we can make fp point to next by writing either of the following two statements:

```
// these two statements are equivalent
fp = &next;
fp = next;  // next() or next(10) will give compiler error.
```

Similarly, if we have an int variable named i, we can use fp to call next, and thereby increment i, by writing either

```
i = (*fp)(i);
i = fp(i);
```

Another example

```
bool is_negative(int n)
{
    return n < 0;
}
```

and we use find_if to locate the first negative element in a vector<int> named v:

```
vector<int>::iterator i = find_if(v.begin(), v.end(), is_negative);
```

We cannot provide default arguments in function pointer like this,

```
int (*fp)(int n = 0); //Compilation error 'fp' : default-arguments are not allowed on this symbol
```

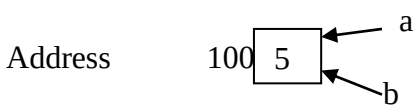
- ## Reference variable

❖ Reference basics

When a variable is declared as **reference**, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

```
int a = 5;
int& b = a;
```

Memory map:



No separate memory

e.g .

```
int main()
{
  int x = 10;
  // ref is a reference to x.
  int& ref = x;
  // Value of x is now changed to 20
  ref = 20;
  cout << "x = " << x << endl ;

  // Value of x is now changed to 30
  x = 30;
  cout << "ref = " << ref << endl ;

  return 0;
}
```

Output:

x= 20

Ref = 30

- **References cannot be NULL**. Pointers are often made **NULL** to indicate that they are not pointing to any valid thing.
- A reference **must be initialized** when declared. There is no such restriction with pointers.
- Can not have reference to a reference , also can not have pointer to a reference.

References are safer and easier to use:

- **Safer**: Since references **must be initialized**, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)
- **Easier to use**: References don't need dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator (->) is needed to access members.

- It is because ++a returns an lvalue, which is basically a reference to the variable to which we can further assign — just like an ordinary variable. It could also be assigned to a reference as follows:

```
int &ref = ++a; // valid

int &ref = a++; // invalid
```

std::ref & std::cref

- std::ref is a standard function that returns a std::reference_wrapper on its argument. In the same idea, std::cref returns std::reference_wrapper to a const reference.
- We can have reference to pointer

```
int* ptr = new int(10);
int& ref = *ptr;
ref = 20; //Both ref & *ptr is now 20
```

Can references refer to invalid location in C++?

Yes

- **Reference to value at uninitialized pointer.**

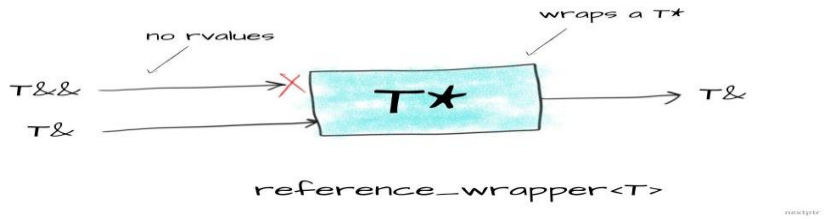
```
int* ptr;
int& ref = *ptr;  // Reference to value at some random memory location
```

- **Reference to a local variable is returned.**

```
int& fun()
{
    int a = 10;
    return a;
}
```

C++ Difference between std::ref(T) and T&?

- This returns a reference_wrapper and not a direct reference to x (ie T&). This reference_wrapper (ie r) instead holds T&.



e.g. 1:

```
template<typename N>
void change(N n) {
    //if n is std::reference_wrapper<int>,
    // it implicitly converts to int& here.
    n += 1;
}

int main() {
    int x = 10;

    int& xref = x;
    change(xref); //Passed by value
    //x is still 10
    std::cout << x << "\n"; //10

    //Be explicit to pass by reference
    change<int&>(x);
    //x is 11 now
    std::cout << x << "\n"; //11

    //Or use std::ref
    change(std::ref(x)); //Passed by reference
    //x is 12 now
    std::cout << x << "\n"; //12
    return 0
}
```

2.

```
void update(int& data)  //expects a reference to int
{
    data = 15;
}
int main()
```

```
{
    int data = 10;

    // This doesn't compile as the data value is copied when its reference is expected.
    //std::thread t1(update, data);

    std::thread t1(update, std::ref(data));  // works
       // Here we need to pass rval only as thread internally called invoke(_STD move())

    t1.join();
    return 0;
}
```

• Static variables

- Static members is remains in memory until the end of the program.
- **File scope**: it can be seen only within a file where it's defined.
- **Initialization** done only once.
- **Visibility**: if it is defined within a function/block, it's scope is limited to the function/block. It cannot be accessed outside of the function/block.
- **Class**: static members exist as members of the class rather than as an instance in each object of the class. So, this keyword is not available in a static member function. Such functions may access only static data members. There is only a single instance of each static data member for the entire class:
  A static data member : class variable
  A non-static data member : instance variable
- Static member function: it can only access static member data, or other static member functions

- Suppose following example of static

```
class Car
{
private:
       static int id;
public:
        Car()
       {
              id++;
       };
};

int Car::id = 100;

Car c1; // id = 101
Car c2; // id = 102
Car c3; // id = 103
```

- Suppose another example of static

```
class A
{
public:
    A() { cout << "A's Constructor Called " << endl;  }
};

class B
{
    static A a;
public:
```

```
    B() { cout << "B's Constructor Called " << endl; }
};

int main()
{
    B b;
    return 0;
}
```

O/ P: The above program calls only B++'s constructor, it doesn't call A's constructor. The reason for this is simple, *static members are only declared in class declaration, not defined.*

```
A B::a;  // definition of a outside class
```

- Suppose another example of static

```
static int fs;
int main()
{
    fs= 1;
}

//Another file
extern int fs;
void func()
{
      fs = 100;//Linker error. Wont available outside the class.
}
```

- Static varaiable in template function

```
template<typename T>
void print(const T x)
{
    static int var = 10;
    cout<<++var;
}
int main()
{
    print(5);
    print(6);
    print('c');
    print(2.0);
}
O/P:
11
12
11
11
```

Here three diff print function gets created for int, float and chat type and three diff static var for these function because static var scope is limited to the function where it is defined. We called int version twice so the var gets increased two times here. This aplies to static variable in template class.

• Static class

- Static class initialization done with constructor call.

```
class X
{
    int m_data;
public:
```

```
    X(int data)
    {
        m_data = data;
    }
};

int main()
{
    static X x1(4); //Calls static class initialization
}
```

- Static object destructor gets called when main exits or when lib function is explicitly called.
- Not called after abort() lib function.

• C++ Specifier

❖ C++ final specifier

- Sometimes you don't want to allow derived class to override the base class' virtual function.

```
class Base
{
public:
    virtual void myfun() final // Check final keyword here
    {
        cout << "myfun() in Base";
    }
};
class Derived : public Base
{
    void myfun()
    {
        cout << "myfun() in Derived\n";
    }
};

int main()
{
    Derived d;
    Base &b = d;
    b.myfun();
    return 0;
}
```

O/P:

error: overriding final function

- final specifier in C++ 11 can also be used to prevent inheritance of class / struct

```
#include <iostream>
class Base final // Check final keyword here
{
};

class Derived : public Base
{
```

```
};

int main()
{
    Derived d;
    return 0;
}
```

O/P:

```
error: cannot derive from ‘final’ base
```

❖ C++ override keyword

It shows the reader of the code that "this is a **virtual** method, that is **overriding** a virtual method of the base class."

- Good for testing.
- Compile time checking can be performed.

❖ Explicitly Defaulted and Deleted Functions in C++ 11

Default:

This makes the compiler generate the default implementations for explicitly defaulted functions.

e.g

```
// Using the default specifier to instruct the compiler
// to create the default implementation of the constructor.
A() = default;
```

What are constrains with making functions defaulted?

A defaulted function needs to be a special member function (default constructor, copy constructor, destructor etc), or has no default arguments.

e.g.

```
// Error, func is not a special member function.
int func() = default;

// Error, constructor B(int, int) is not a special member function.
B(int, int) = default;

// Error, constructor B(int=0) has a default argument.
B(int = 0) = default;
```

Delete:

operator **delete** had only one purpose, to deallocate a memory that has been allocated dynamically.

e.g:

```
// Delete the copy constructor
A(const A&) = delete;

// Delete the copy assignment operator
```

```
A& operator=(const A&) = delete;
```

❖ How to assign any object to premitive data type (int, float...)

Ans : Need to define operator int () to allow the conversion from class to int data type

```
 class Base()
 {
      int var;
      public:
      Base(int val) : var(val) {}
      operator int() const { return var;} // Note:There is not return type
 };
 int main()
 {
      Base b(100);
      int tmp = b;
      cout<<tmp;
 }
O/P:
100
```

❖ How to call any function before main function

Ans: Just create an global class type and from the constructor you can call any function you want.

```
void fun() { cout<< "Inside fun";}
class Base()
 {
      public:
      Base(int val) : var(val) { fun()}
 };
 Base b;
 int main()
 {
 }
```

❖ How to print anthing without using loop or recursion

```
void fun() { cout<< "Inside fun";}
class Base()
 {
      public:
      Base(int val) : var(val) { cout<< "Inside fun";}
 };
 int main()
 {
      Base b[10];
 }
```

❖ How to call destructor explicitly.

```
int main()
 {
      Base obj;
      obj.~Base();
 }

 O/P:
 Constructor
```

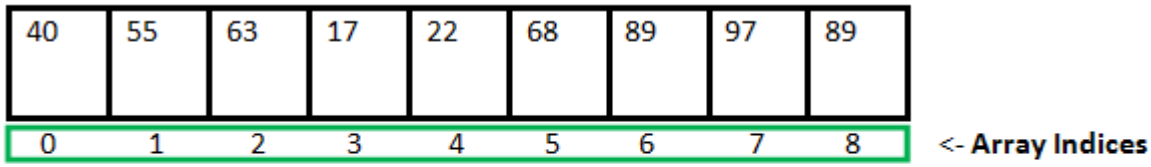
```
Destructor  //Called explicitly
Destructor  //Called when object gets out of scope
```

Data Structures

- **Arrays in C/C++**

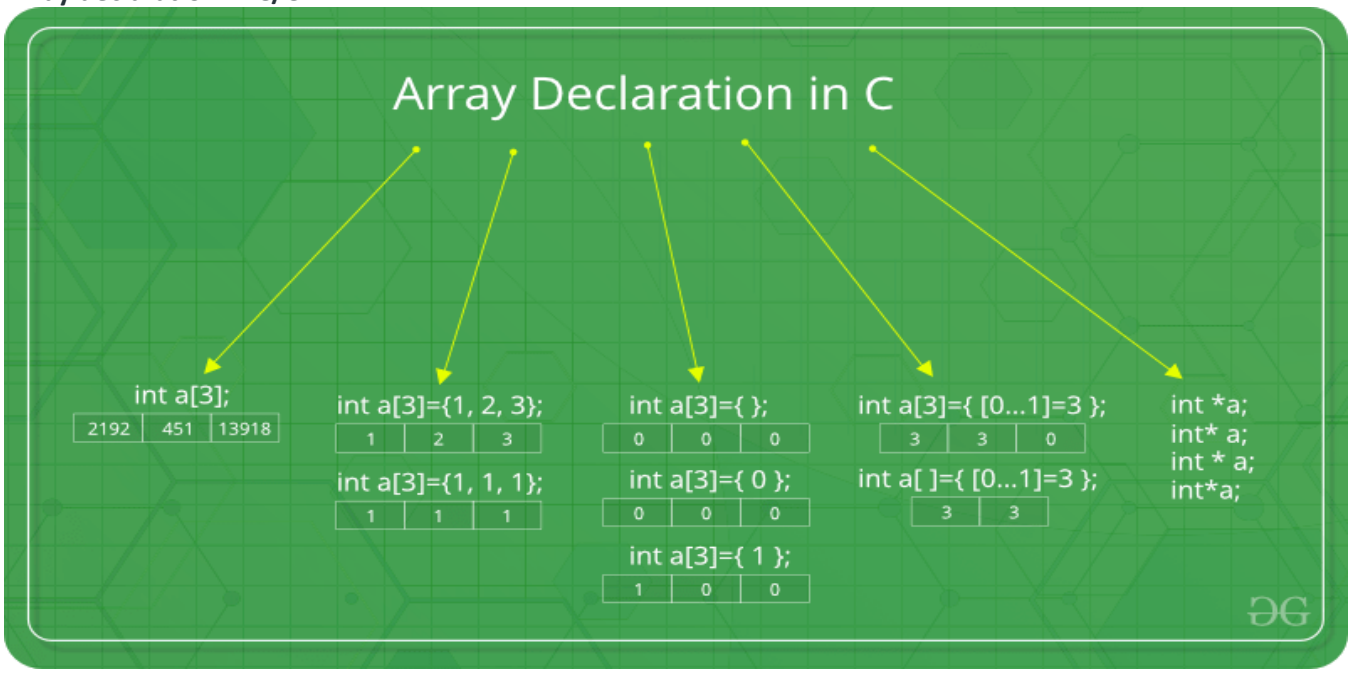
❖ **Array basics:**

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.



Array Length = 9
First Index = 0
Last Index = 8

Array declaration in C/C++:



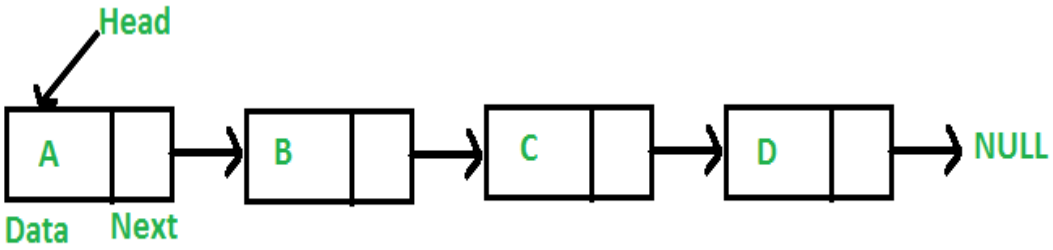
Main issue with the array:
No Index Out of bound Checking:
There is no index out of bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

```
// This C program compiles fine as index out of bound is not checked in C.

#include <stdio.h>
int main()
{
    int arr[2];

    printf("%d ", arr[3]);
```

```
    printf("%d ", arr[-2]);

    return 0;
}
```

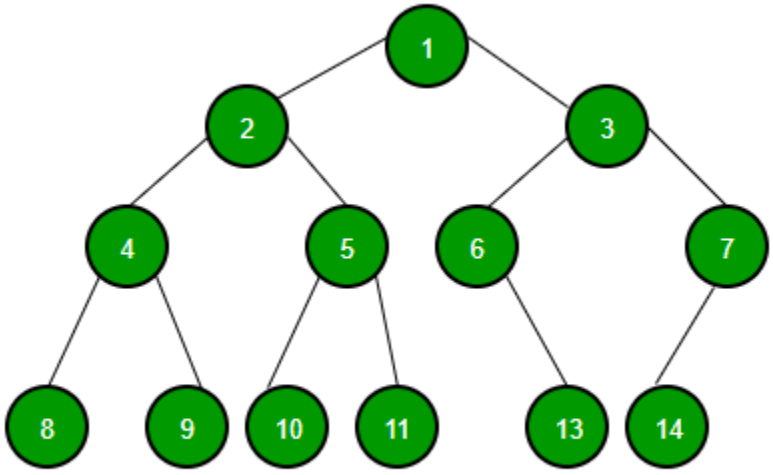
- ## Linked List Data Structure

A linked list is a linear data structure, in which the elements are ***not stored at contiguous memory*** locations. The elements in a linked list are ***linked using pointers*** as shown in the below image:



- ## Binary Tree Data Structure

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

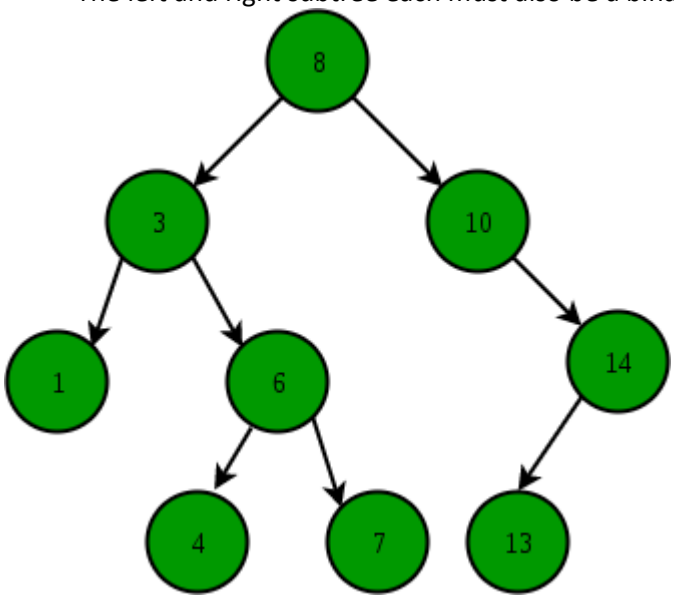
- Function to find sum of all the elements

```
int addBT(Node* root)
{
    if (root == NULL)
        return 0;
    return (root->key + addBT(root->left) + addBT(root->right));
}
```

• Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

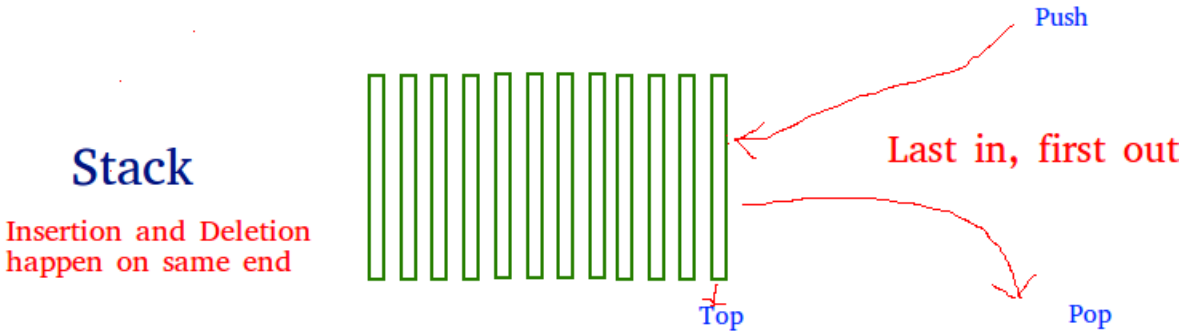


Advantages of BST over Hash Table

- We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.
- Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.
- BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.
- With Self-Balancing BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.

- ## Stack Data Structure

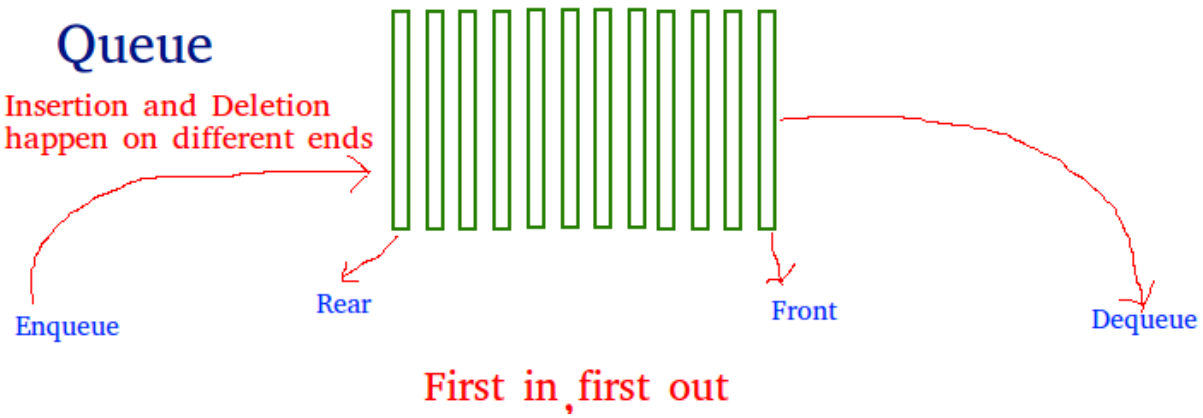
- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be **LIFO**(Last In First Out) or **FILO**(First In Last Out).



There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

- ## Queue Data Structure

- A Queue is a linear structure which follows a particular order in which the operations are performed. The order is **First In First Out** (**FIFO**).
- A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

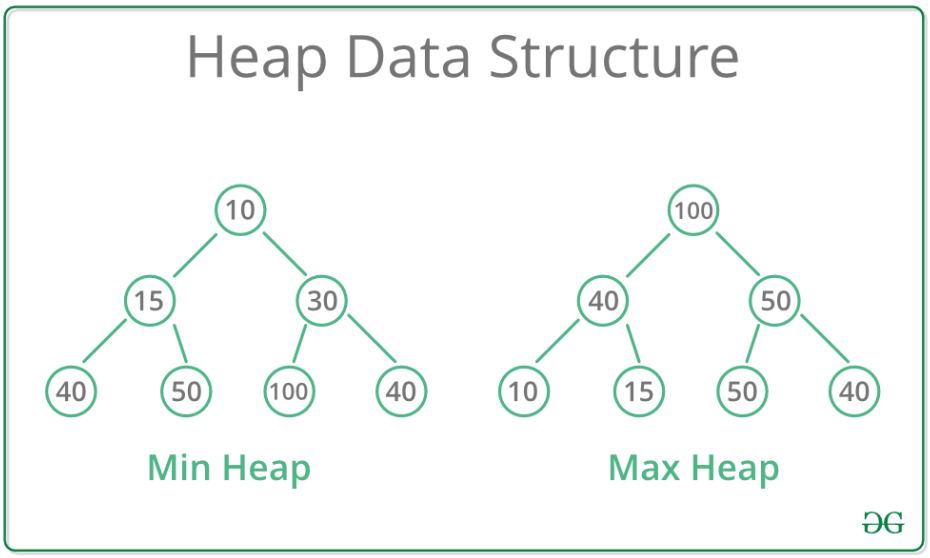


- ## Heap Data Structure

- A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

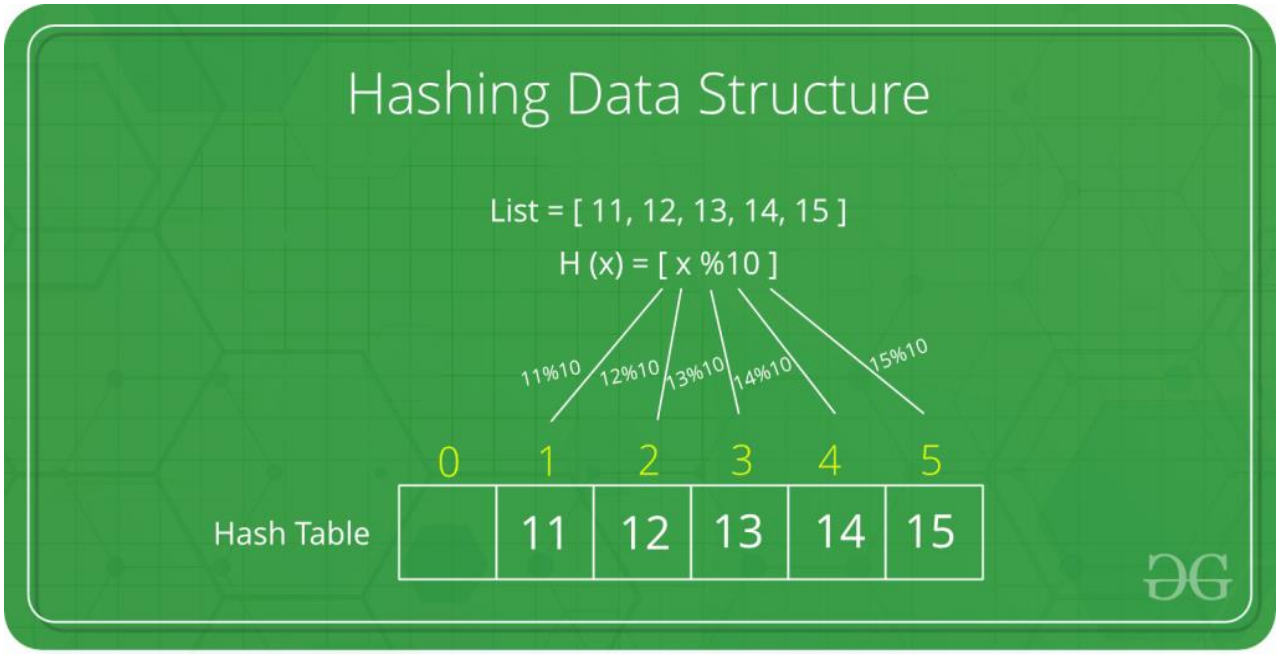
1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



- ## Hashing Data Structure

- Hashing is a technique or **process of mapping keys**, **values into the hash table** by using a **hash function**. It is done for faster access to elements. The efficiency of mapping depends on the **efficiency of the hash function used.**
- Let a hash function H(x) maps the value at the index x%10 in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

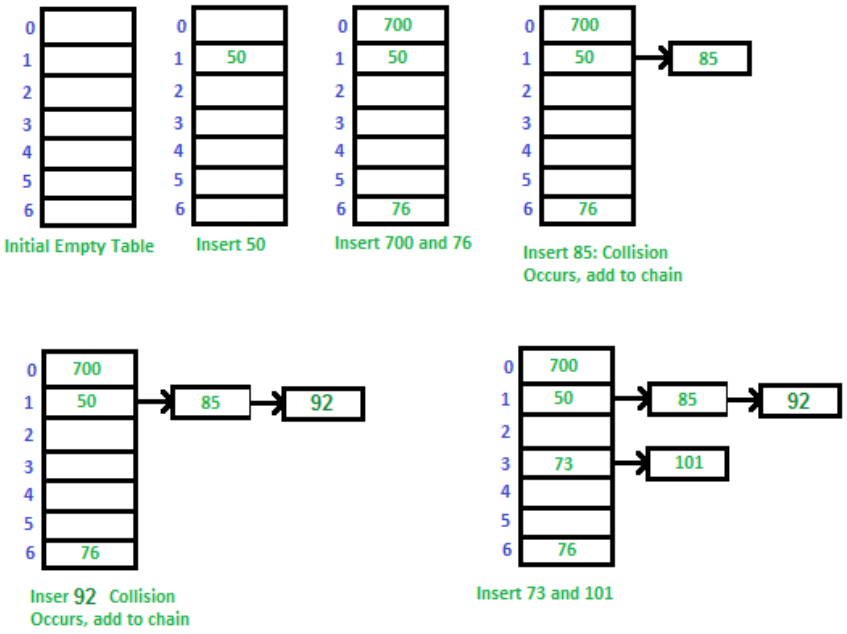
How to handle Collisions?

There are mainly two methods to handle collision:

1) Separate Chaining

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Advantages:

1) Simple to implement.
2) Hash table never fills up, we can always add more elements to the chain.
3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
2) Wastage of Space (Some Parts of hash table are never used)
3) If the chain becomes long, then search time can become O(n) in the worst case.
4) Uses extra space for links.

2)Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

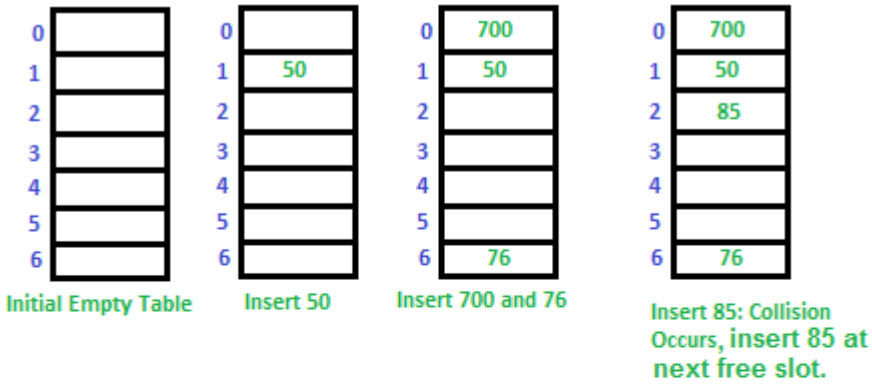
Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

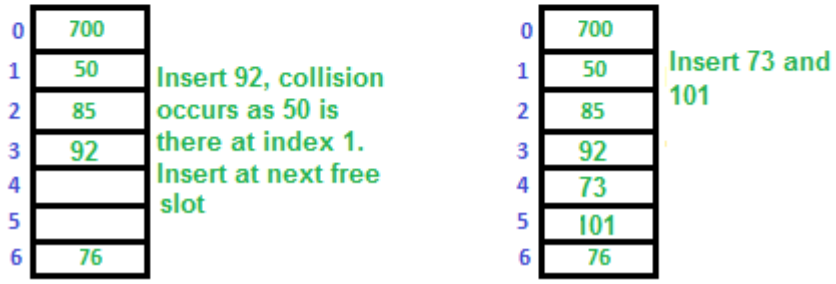
Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".
The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.
Open Addressing is done in the following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1 as seen in the example below.
Let **hash(x)** be the slot index computed using a hash function and **S** be the table size

```
If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
```

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.





Challenges in Linear Probing :

1. **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
2. **Secondary Clustering:** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

b) Quadratic Probing We look for i^2'th slot in i'th iteration.

```
let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*2) % S

If (hash(x) + 1*2) % S is also full, then we try (hash(x) + 2*2) % S

If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*2) % S

.................................................

.................................................
```

c) Double Hashing We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

```
let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S
```

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Design patterns

- Design Patrtern is general **repeatable solution to a commonly occurring problem** in software design.
- They give a developer a tried and tested solution to work with.
- They are language neutral.
- They are proven track record as they are widely used and thus reducing technical risk.

❖ Singleton design pattern (Creational)

- It is a **creational** design pattern.
- Used where only one instance of an object is needed throught the lifetime of an application.
- Singleton class is instantiated at the time of first access & same is used thereafter.
- E.g. Office of an princilple (Office is same principle may change).
- Used to control access to resources such as database connection or sockets. (E.g License of one connection to database, Logger Class)

```
class Singleton
{
private:
	static Singleton* m_single;
	Singleton() { }
public:
	static Singleton* getInstance();
	void myMethod();
};

Singleton * Singleton::m_single = NULL;

Singleton* Singleton::getInstance()
{
	if(! m_single)
	{
		m_single = new Singleton();
	}
	return m_single;
}

void Singleton::myMethod()
{
	cout << "Method of singleton";
}
```

- General purpose of singleton pattern is to limit the number of instance of class to one however patern can be extended by many ways to control the number of instance.

```
class CMyClass
{
private:
	static int nCount;
	static int maxCount;
	CMyClass() { }
public:
	static CMyClass* createInstance();
	void myMethod();
};

int CMyClass::nCount = 0;
int CMyClass::maxCount = 3;
```

```
CMyClass* CMyClass::createInstance()
{
	CMyClass* ptr = NULL;
	if( maxCount > nCount)
	{
		ptr = new CMyClass();
		++nCount;
	}
	return ptr;
}

void CMyClass::myMethod()
{
	cout << "Method of singleton";
}
```

- **Note** : If maxCount =1 then this is Singleton class.

- **thread safe singleton c++**

Using double-checked locking

```
Singleton * Singleton::Instance() {
	if(instance == nullptr) {
		lock_guard<mutex> lock(m_);
		if(instance == nullptr) {
			instance = new Singleton();
		}
	}
	return instance;
}
```

- **Singleton Pattern Versus Static Class**

- While a ***static class allows only static methods*** and and ***you cannot pass static class as parameter***.
- A Singleton can implement interfaces, inherit from other classes and allow inheritance. While a static class cannot inherit their instance members.
- ***Singleton Objects stored on heap*** while ***static class stored in stack***.
- Singleton Objects can have constructor while Static Class cannot.
- ***Singleton Objects can dispose*** but not static class.
- Singleton Objects can clone but not with static class.

```
Singleton* s = Singleton::getInstance();
delete s; // We can delete singleton instance after use.
s = nullptr;
s = Singleton::getInstance();
```

❖ Factory design pattern (Creational)

- It creates object for you, rather you initiating object directly.
- Also known as virtual constructor.

```
// A design without factory pattern
#include <iostream>
using namespace std;
```

```
// Library classes
class Vehicle {
public:
    virtual void printVehicle() = 0;
};
class TwoWheeler : public Vehicle {
public:
    void printVehicle()  {
        cout << "I am two wheeler" << endl;
    }
};
class FourWheeler : public Vehicle {
    public:
    void printVehicle()  {
        cout << "I am four wheeler" << endl;
    }
};
// Client (or user) class
class VehicleFactory {
public:
    VehicleFactory (int type)  {

        // Client explicitly creates classes according to type
        if (type == 1)
            pVehicle = new TwoWheeler();
        else if (type == 2)
            pVehicle = new FourWheeler();
        else
            pVehicle = NULL;
    }
     // We can write creation logic either in constructor or in
     // static function like this.
     static Vehicle* getVehicleObj(int type)
     {
          if (type == 1)
            pVehicle = new TwoWheeler();
        else if (type == 2)
            pVehicle = new FourWheeler();
        else
            pVehicle = NULL;

        // We can do preprocessing on the created table
          pVehicle->applyLabel();
          return pVehicle;
     }

    ~ VehicleFactory ()   {
        if (pVehicle)
        {
            delete[] pVehicle;
            pVehicle = NULL;
        }
    }
```

```
    Vehicle* getVehicle() {
        return pVehicle;
    }
private:
    Vehicle *pVehicle;
};

// Driver program
int main()
{
   // We can get object by constructor like
    VehiclFactory *pClient = new VehiclFactory(1);
    Vehicle * pVehicle = pClient->getVehicle();

    // We can get object by static function like
    Vehicle * pVehicle=  VehiclFactory::getVehicleObj(1);

    pVehicle->printVehicle();
    return 0;
}
```

Advantages:

- Less code chage if We can object creation process.
- We create object without exposing creation logic to the client.
- We get benefits of virtual constructor.

❖ Abstract Factory Pattern(Creational)

Introduction

- Abstract Factory design pattern is one of the Creational pattern.
- Abstract Factory pattern is almost similar to Factory Pattern is considered as **another layer of abstraction over factory pattern.**
- Abstract Factory patterns work around a **super-factory which creates other factories.**

❖ Builder Design Pattern(Creational)

- Builder pattern aims to "Separate the construction of a complex object from its representation so that the same construction process can create different representations."
- It is used to construct a complex object step by step and the final step will return the object.
- The process of constructing an object should be generic so that it can be used to create different representations of the same object.
- **Usage examples:** The Builder pattern is a well-known pattern in C++ world. It's especially useful when you need to create an object with lots of possible configuration options.

e.g:

Consider the ***construction of a home. Home is the final end product (object) that is to be returned as the output of the construction process.*** It will have many steps like basement construction, wall construction, and so on roof construction. Finally, the whole home object is returned. Here using the same process you can build houses with different properties.

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

```
class PersonBuilder;

class Person
{
    std::string m_name, m_street_address, m_city;  // Personal Detail
    Person(std::string name) : m_name(name) {}

public:
    friend class PersonBuilder;
    friend ostream& operator<<(ostream& os, const Person& obj);
    static PersonBuilder create(std::string name);
};



ostream& operator<<(ostream& os, const Person& obj)
{
    return os << obj.m_name
        << std::endl
        << "lives : " << std::endl
        << "at " << obj.m_street_address
        << " in " << obj.m_city
        << std::endl;
}

class PersonBuilder
{
    Person person;

public:
    PersonBuilder(string name) : person(name) {}

    operator Person() const { return move(person); }

    PersonBuilder& lives() { return *this; }
    PersonBuilder& at(std::string street_address) {
        person.m_street_address = street_address;
        return *this;
    }
    PersonBuilder& in(std::string city) {
        person.m_city = city;
        return *this;
    }
};

PersonBuilder Person::create(string name) { return PersonBuilder{ name }; }


int main()
{

    Person p = Person::create("John")
        .lives()
        .at("123 London Road")
        .in("London");

   cout << p << endl;
}
```

Benefits of Builder Design Pattern

- The number of lines of code increases at least to double in builder pattern. But the effort pays off in

terms of design flexibility, fewer or no parameters to the constructor and much more readable code.

- Builder Design Pattern also helps in minimizing the number of parameters in constructor & thus there is no need to pass in null for optional parameters to the constructor.
- Immutable objects can be built without much complex logic in the object building process.

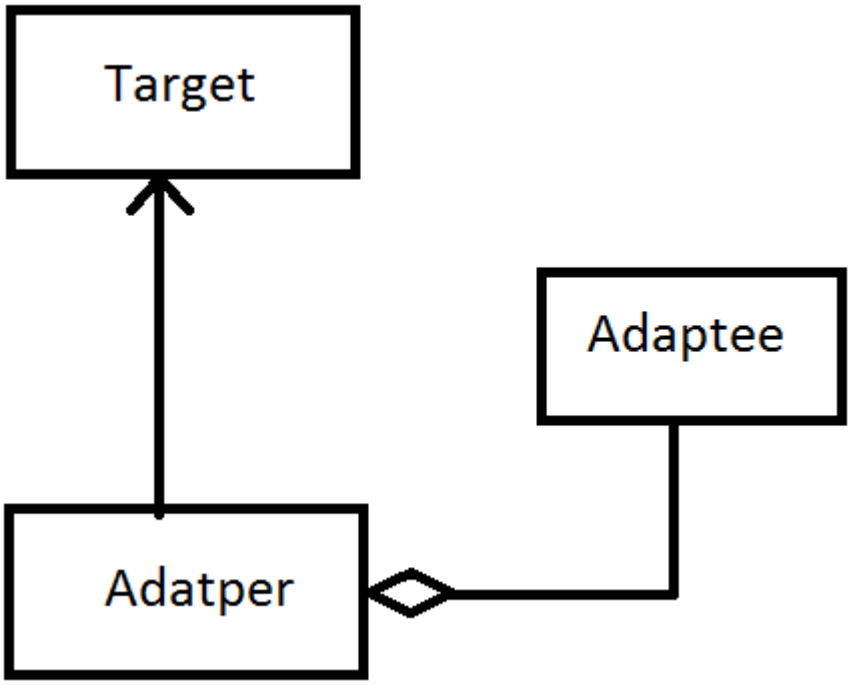
❖ Prototype Design Pattern(Creational)

```
/** Bullet is the base Prototype */
class Bullet
{
protected:
    string _bulletName;
    float _speed;
    float _firePower;
    float _damagePower;
    float _direction;

public:
    Bullet() {}
    Bullet(string bulletName, float speed, float firePower, float damagePower)
        : _bulletName(bulletName), _speed(speed), _firePower(firePower),
_damagePower(damagePower)
    {}
    virtual ~Bullet() {}
    virtual unique_ptr<Bullet> clone() = 0;
    void fire(float direction)
    {
        _direction = direction;

        cout << "Name        : " << _bulletName << endl
            << "Speed       : " << _speed << endl
            << "FirePower   : " << _firePower << endl
            << "DamagePower : " << _damagePower << endl
            << "Direction   : " << _direction << endl << endl;
    }
};

/** SimpleBullet is a Concrete Prototype */
class SimpleBullet : public Bullet
{

public:
    SimpleBullet(string bulletName, float speed, float firePower, float damagePower) :
        Bullet(bulletName, speed, firePower, damagePower)
    {
    }

    unique_ptr<Bullet> clone() override
    {
        return make_unique<SimpleBullet>(*this);
    }
};

/** GoodBullet is the Concrete Prototype */
class GoodBullet : public Bullet
{

public:
    GoodBullet(string bulletName, float speed, float firePower, float damagePower)
```

```
        : Bullet(bulletName, speed, firePower, damagePower)
    {
    }

    unique_ptr<Bullet> clone() override
    {
        return std::make_unique<GoodBullet>(*this);
    }
};


/** Opaque Bullet type, avoids exposing concrete implementations */
enum BulletType
{
    SIMPLE,
    GOOD
};

/** BulletFactory is the client */
class BulletFactory
{
private:
    unordered_map<BulletType, unique_ptr<Bullet>, hash<int> > m_Bullets;

public:
    BulletFactory()
    {
        m_Bullets[SIMPLE] = make_unique<SimpleBullet>("Simple Bullet", 50, 75, 75);
        m_Bullets[GOOD] = make_unique<GoodBullet>("Good Bullet", 75, 100, 100);
    }

    unique_ptr<Bullet> createBullet(BulletType BulletType)
    {
        return m_Bullets[BulletType]->clone();
    }
};

int main()
{

    BulletFactory bulletFactory;

    auto Bullet = bulletFactory.createBullet(SIMPLE);
    Bullet->fire(90);

    Bullet = bulletFactory.createBullet(GOOD);
    Bullet->fire(100);
}
```

❖ Adapter design pattern (Structural)

- Convert **the interface of a class into another interface** clients expect.
- Wrap an existing class with a new interface.
- Adapter pattern works as a bridge between two incompatible interfaces.



E.g.

```
class Rectangle
{
  public:
    virtual void draw() = 0;
};

// Legacy component
class LegacyRectangle
{
  public:
     LegacyRectangle(Coordinate x1, Coordinate y1, Coordinate x2,
     Coordinate y2){}
};

// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
  public:
    RectangleAdapter(Coordinate x, Coordinate y, Dimension w, Dimension
h): LegacyRectangle(x, y, x + w, y + h) {
      }
};

int main()
{
  Rectangle *r = new RectangleAdapter(120, 200, 60, 40);
  r->draw();
}
```

❖ Proxy Design Pattern (Structural)

- Proxy means **'in place of'**, representing' or 'in place of' **or 'on behalf of'** are literal meanings of proxy and that directly explains **Proxy Design Pattern**.
- A real world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required.

e.g:

```
#include <iostream>
using namespace std;
class Subject
{
public:
    virtual void request() = 0;
    virtual ~Subject() {}
};

class RealSubject : public Subject
{
public:
    void request() {
        cout << "RealSubject.request()" << endl;
    }
};

class Proxy : public Subject
{
private:
    Subject* realSubject;
public:
    Proxy() : realSubject(new RealSubject())
    {}
    ~Proxy() {
        delete realSubject;
    }
    // Forward calls to the RealSubject:
    void request() {
        realSubject->request();
    }
};

int main() {
    Proxy p;
    p.request();
   }
```

When to use this pattern?

- Proxy pattern is used when we need to create a wrapper to cover the main object's complexity from the client.

Interesting points:

- There are few differences between the related patterns. Like **Adapter pattern gives a different interface** to its subject, while **Proxy patterns provides the same interface** from the original object but the decorator provides an enhanced interface. **Decorator pattern adds additional behaviour** at runtime.

❖ Decorator Design Pattern (Structural)

- Attach **additional responsibilities** to an object dynamically.
- Decorators provide flexible alternatives to subclassing for extending functionality.

e.g:

```
class Computer
{
public:
    virtual void display()
    {
        cout << "I am a computer..." << endl;
    }
};

class CDDrive : public Computer
{
private:
    Computer* c;
public:
    CDDrive(Computer* _c)
    {
        c = _c;
    }
    void display()
    {
        c->display();
        cout << "with a CD Drive..." << endl;
    }
};

class Printer : public Computer
{
private:
    CDDrive* d;
public:
    Printer(CDDrive* _d)
    {
        d = _d;
    }
    void display()
    {
        d->display();
        cout << "with a printer..." << endl;
    }
};

int main()
{
    Computer* c = new Computer();
    CDDrive* d = new CDDrive(c);
    Printer* p = new Printer(d);

    p->display();
}
```

e.g 2:

```
class Window
{
public:
    virtual void draw() = 0;
    virtual string getDescription() = 0;
    virtual ~Window() {}
};

class SimpleWindow : public Window
{
public:
    void draw() {
        // draw window
    }
    string getDescription() {
        return "simple window\n";
    }
};

class WindowDecorator : public Window
{
protected:
    Window* m_decoratedWindow;
public:
    WindowDecorator(Window* decoratedWindow) :
        m_decoratedWindow(decoratedWindow) {}
    string getDescription() {
        return m_decoratedWindow->getDescription() + "with scrollbars\n";
    }
};

int main()
{
    Window* simple = new SimpleWindow();
    cout << simple->getDescription() << endl;

    Window* simple = new WindowDecorator();
    cout << simple->getDescription() << endl;
}
```

O/P:

simple window

simple window with scrollbars. //Decorator output

When to use the Decorator Design Pattern?

- Employ the Decorator Design Pattern when you need to be able to assign extra behaviours to objects at runtime without breaking the code that uses these objects.

Difference between Adapter & Decorator Design Pattern?

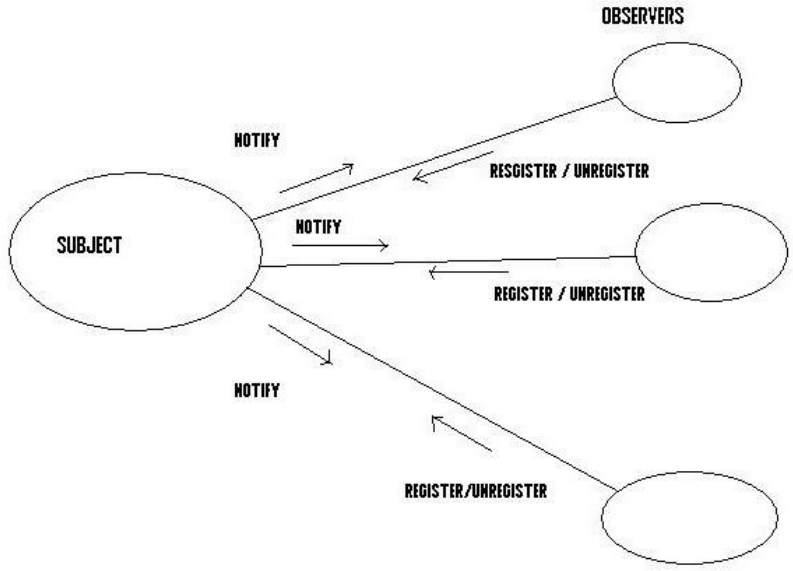
- Adapter changes the interface of an existing object.
- Decorator enhances the interface of an existing object

Difference between Proxy & Decorator Design Pattern?

- Proxy provides a somewhat same or easy interface.
- Decorator provides enhanced interface

❖ Observer design pattern (Behavior)

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- The "View" part of Model-View-Controller.



E.g:

```
class Subject {
    // 1. "independent" functionality
    vector < class Observer * > views; // 3. Coupled only to "interface"
    int value;
  public:
    void attach(Observer *obs) {
        views.push_back(obs);
    }
    void setVal(int val) {
        value = val;
        notify();
    }
    int getVal() {
        return value;
    }
    void notify();
};

class Observer {
    // 2. "dependent" functionality
    Subject *model;
    int denom;
  public:
    Observer(Subject *mod, int div) {
        model = mod;
```

```
        denom = div;
        // 4. Observers register themselves with the Subject
        model->attach(this);
    }
    virtual void update() = 0;
  protected:
    Subject *getSubject() {
        return model;
    }
    int getDivisor() {
        return denom;
    }
};

void Subject::notify() {
  // 5. Publisher broadcasts
  for (int i = 0; i < views.size(); i++)
    views[i]->update();
}

class DivObserver: public Observer {
  public:
    DivObserver(Subject *mod, int div): Observer(mod, div){}
    void update() {
        // 6. "Pull" information of interest
        int v = getSubject()->getVal(), d = getDivisor();
        cout << v << " div " << d << " is " << v / d << '\n';
    }
};

class ModObserver: public Observer {
  public:
    ModObserver(Subject *mod, int div): Observer(mod, div){}
    void update() {
        int v = getSubject()->getVal(), d = getDivisor();
        cout << v << " mod " << d << " is " << v % d << '\n';
    }
};

int main() {
  Subject subj;
  DivObserver divObs1(&subj, 4); // 7. Client configures the number and
  DivObserver divObs2(&subj, 3); //    type of Observers
  ModObserver modObs3(&subj, 3);
  subj.setVal(14);
}
```

C++ Special

❖ Some interesting facts about string in C++

- **What is difference between char* & char[]**

```
char * str = “Hello world”;
str[0] = 'N'; // Ilegal
```

This will place “Hello world” in read only part of the memory & making ‘str’ a pointer to that. Makes any writing operation on this memory illegal.
This But if memory is created either on stack or on heap like,

```
char * str = new char (20); // str is created on Heap
strcpy_s(str, 20, " Hello world");
str[0] = 'N'; // Legal


char str[10] = "My String";// str is created on stack
str[0] = 'N'; // Legal
```

This puts literal string in ***read only memory*** & copies the string to newly allocated memory on the heap/stack thus making operation on it legal.

❖ What’s difference between char s[] and char *s in C?

Consider below two statements in C. What is the difference between the two?

```
char s[] = "geeksquiz";

char *s  = "geeksquiz";
```

Below are the key differences:

Char a[10] = "geek";	Char *p = "geek";
1) a is an array	1) p is a pointer variable
2) sizeof(a) = 10 bytes	2) sizeof(p) = 4 bytes
3) a and &a are same	3) p and &p aren't same
4) geek is stored in stack section of memory	4) p is stored at stack but geek is stored at code section of memory
5) char a[10] = "geek"; a = "hello"; //invalid > a, itself being an address and string constant is also an address, so not possible.	5) char *p = "geek"; p = "india"; //valid
6) a++ is invalid	6) p++ is valid
7) char a[10] = "geek"; a[0] = 'b'; //valid	7) char *p = "geek"; p[0] = 'k'; //invalid > Code section is r- only.

The statements '**char s[] = "geeksquiz"**' creates a character array which is like any other array and we can do all array operations. The only special thing about this array is, although we have initialized it with 9 elements, its size is 10 (Compiler automatically adds '\0')

- **How to reverse a string in C++.**

```
void myStrRev(char* str)
{
    int len = strnlen_s(str, 100);
    for (int i=0, j=len-1; i<len-1, j>i; i++, j--)
    {
          char s =str[i];
          str[i]=str[j];
          str[j]=s;
    }
}
```

Here we are dealing with actual memory locations so the reverse string is propagated even after function got over. This applies to int array also. Since we are accessing array[] (Particular element address) actual copy gets modified.

❖ Name mangling

- C compiler links a function or symbol with appending underscore before it. e.g. function main() links to Symbol _main() & variable int count to _count.
- C++ is having function overloading mechanism. If c++ uses c mechanism to link functions, the purpose of overloading will not be resolved.
- To overcome this limitation C++ linker uses name mangling mechanism.
- It is a mechanism of C++ compiler to link functions & variables of same name to resolve them to some unique symbol names by altering the names with some extra info like index, argument size etc.
- Function name mangling is done as
- <function index> <function name> @<argument size>
- And variable name mangling is done as
- ?<variable name> @@ <Unique Id>

e.g.

- int add (int a, int b) links to _1add@8
- int add(int a, int b, int c) links to _2add@12
- int extern_var links to ? extern_var @@3HA(Unique id)

❖ How delete[] know how many objects to delete?

- Ans:

1. Over allocation

```
class Base{
public:
int b_var;
};

int main()
{
      Base * bp = new Base[n];
      /*char*  ptr = (char*) operator new[](WORDSIZE + n*sizeof(Base));
      Base * bp = (Base*)(tmp+WORDSIZE);
      *(size_t*) tmp = n;
      for(int i = 0; i<n; i++)
```

```
            new(p+i)Base();*/ // Placement new

        delete[] bp;
        /*size_t n = *(size_t*)((char*)p - WORDSIZE);
        while(n-- != 0)
              (p+n)->~Base();
        operator delete[]((char*)p - WORDSIZE)*/
}
```

2.Associative array

❖ Some interesting facts about static member functions in C++

- static member functions do not have this pointer.

e.g.

```
static Test * fun() {
    return this; // compiler error
  }
```

- A static member function cannot be virtual.

e.g.

```
static virtual Test * fun() {}// compiler error
```

- Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.

e.g.

```
static void fun() {}
void fun() {} // compiler error
```

- A static member function cannot be declared const, volatile, or const volatile.

e.g.

```
static void fun() const { } // compiler error
```

• Default arguments

- When we give a parameter a default argument, we're saying that callers can omit that argument if they wish. If they supply an argument, the function will use it. If they omit the argument, the compiler will substitute the default. Thus, users can call this function in either of two ways.

  e.g.

```
float calc (float s=200, float h=8);

calc();
calc(3000);
calc(3000,6); //Will all works.
```

- All non default parameters must be at left side.
- We cannot provide default arguments in function pointer like this,

  int (*fp)(int = 0) ;

• Inline functions

- When function execution switches from calling function to called function some time is wasted to switch to that function & return back to original function.
- This time is considerable for small size function i.e. of one line or two lines. So C++ recommends such

function are not called their code is substituted which improves speed.
- Pre-processor **macros** look same but they do not have permission to access class member data.
- Any function defined within a class body is automatically inline, but you can also make a non class function inline by preceding it with the inline keyword.

e.g.

```
inline int next(int n)
{
	return n + 1;
}
```

Limitations:
- In some situation compiler can not perform inlining in these cases it simply reverts to the ordinary form of a function by taking the inline definition & creating storage for the function just as it does for non-inline.
- Compiler can not perform inlining if the **function is too complicated.**
- Any **looping** is considered as complicated to expand as an inline.
- Inline is just suggestion to compiler, the compiler is not forced to inline anything at all.

• Function overloading

In c++ possible to define multiple functions with same name provided they differ at least
1. Number of parameters
2. Data type of parameter

e.g.
1.
```
void func(float a, int b);

void func(float a, float b);
void func(float a, float b, int c);
```

2. Function which only differs parameter by constness are not example of overloading.
```
void func(float a, int b);
void func(float a, const int b); // Compilation error
void func(const float a, int b); // Compilation error
```

3. Function which differs parameter by either pointer or reference are example of overloading.

```
void func(float* a, int b);
void func(float& a, int b);
void func(float a, int b);
```

But calling func for float & int makes this call ambiguous as both 2nd & 3rd function can be called. Which results into compilation error.

4. Function which differs parameter by either constness of pointer or constness of reference are example of overloading.

```
void func(float& a, int b);
void func(const float& a, int b);
void func(float* a, int b);
void func(const float* a, int b);
```

5. Function which differs parameter by either static, extern, mutable are not example of overloading.

```
void func(static int* a, int* b);
void func(extern int* a, int* b); // Compilation error
void func(mutable int* a, int* b); // Compilation error
```

6.

```
void f1(int x)
void f1(unsigned x)

in main
f1(-2);  // int function called
f1(-2.5);  // Compilation error
```

- **Operator overloading**

we can make operators to work for user defined classes. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.
e.g:

```
    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;
  }

  Complex a,b,c;
  C = a+b; //a.operator+(b);
```

- Operators which was not allowed to overload.

```
. (dot)
::
?:
sizeof
```

- && The right operand is evaluated only if the left operand is true.
- || The right operand is evaluated only if the left operand is false.
- ++ & -- operator comes in prefix & postfix forms.
- Code for pre increament & post increament operator overloading.

```
//Pre increament
Check& operator ++ ()
{
    this->i = ++i;
    return *(this);
}

//Post increament
```

```
/* Notice int inside barcket which indicates postfix increment. */
Check operator ++ (int)
{
    Check temp = *(this);
    this->i = i++;
    return temp;
}
```

- A example of class with operator overloading

```
class Obj
{
    int val;
public:
    Obj() //Default Constructor
    {
        val = 0;
    }

    Obj(int val) //Parameterized Constructor
    {
        this->val = val;
    }

    Obj operator+ (const Obj& o) // + operator overloading
    {
        return Obj(val + o.val);
    }

    void operator= (const Obj& o) // = operator overloading
    {
        if (this != &o)
        {
            this->val = o.val;
        }
    }

    Obj operator++ (int) // ++ (Post) operator overloading
    {
        Obj o = *(this);
        this->val++;
        return o;
    }

    Obj& operator++ () // ++ (Pre) operator overloading
    {
        this->val++;
        return *(this);
    }

    bool operator&& (const Obj& o) // && operator overloading
    {
        if (val)
        {
            if (o.val)
            {
                return true;
            }
        }
        return false;
    }
```

```
    bool operator|| (const Obj& o) // || operator overloading
    {
        if (!val)
        {
            if (!o.val)
            {
                return false;
            }
        }
        return true;
    }
};
```

why friend function cannot overload assignment operator?
Assignment(=) operator is a special operator that will be provided by the constructor to the class when programmer has not provided(overloaded) as member of the class.(like copy constructor).
When programmer is overloading = operator using friend function, two = operations will exists:
1) compiler is providing = operator
2) programmer is providing(overloading) = operator by friend function.
Then simply ambiguity will be created and compiler will gives error. Its compilation error.

❖ Self assignment check in assignment operator

- In C++, assignment operator should be overloaded with self assignment check.
- If we have an object say a1 of type Array and if we have a line like a1 = a1 somewhere, the program results in unpredictable behavior because there is no self assignment check.
- To avoid the above issue, self assignment check must be there while overloading assignment operator.

e.g:

```
Test& Test::operator = (const Test& rhs)
{
    /* SELF ASSIGNMENT CHECK */
    if (this != &rhs)
    {
        // Deallocate old memory
        delete ptr;
        // allocate new space
        ptr = new int(*rhs.ptr);
    }
    return *this;
}
```

• Global variable

we can access a global variable if we have a local variable with same name using scope resolution operator (::).

e.g.

```
int x;  // Global x

int main()
{
    int x = 10; // Local x
    cout << "Value of global x is " << ::x << endl; // print Global x
    cout << "Value of local x is " << x;   // print Local x
    getchar();
```

```
    return 0;
}
```

• Local (or Automatic) Variables

- The variables are called local to represent the idea that their lifetime is tied to the function where they are declared.

e.g.

```
int Square(int number)
{
    int result;
    result = number * number;
    return result;
}
```

- when the Square() function is called, local storage is allocated for number and result. When the function finally exits, its local storage is deallocated.

Local Copies

- Local parameters are basically local copies of the information from the caller. This is also known as pass by value.

A common bug: a function returning a pointer to a local variable

```
int* local_pointer()
{
    int temp = 100;
    // returns a pointer to the local int
    return(&temp);
}
```

- The problem is that the local int, temp, is allocated only while local_pointer() is running. When local_pointer() exits, all of its locals are deallocated.

• Mutable variable

A mutable field can be changed even in an object accessed through a const pointer or reference, or in a const object, so the compiler knows not to stash it in R/O memory.

What is the need of mutable?

Sometimes there is requirement to modify one or more data members of class / struct through const function even though you don't want the function to update other members of class / struct. This task can be easily performed by using mutable keyword.

```
// PROGRAM 1
#include <iostream>
using std::cout;

class Test {
public:
    int x;
    mutable int y;
    Test() { x = 4; y = 10; }
};
int main()
{
    const Test t1;
    t1.y = 20;
    cout << t1.y;
```

```
    return 0;
}

/* OUTPUT: 20 */

Another example
// to change the place holder
void changePlacedOrder(string p) const
{
    placedorder = p;
    // We can change the mutable variable even in const function.
}
```

• Register variable

- Registers are faster than memory to access, so the variables which are most frequently used in a C++ program can be put in registers using register keyword.
- The keyword register hints to compiler that a given variable can be put in a register.
- It's compiler's choice to put it in a register or not.
- If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid.

E.g .

```
int main()
{
    register int i = 10;
    int* a = &i;
    printf("%d", *a);
    getchar();
    return 0;
}
```

- Register is a storage class, and C doesn't allow multiple storage class specifiers for a variable. So, register can not be used with static .

```
int i = 10;
register static int*a = &i; //Compilation error
```

• Union

- Piles all data into single space.
- sizeof of union is amount of space necessary for largest item you have put in union.

```
union packed
{
      char i;
      short j;
      int k;
      double d;
};
packed x;
x.i = 'c';
cout << x.i;
x.d = 3.14;
cout << x.d;
cout << x.i; // garbage
```

• Dynamic Memory allocation

- Memory allocated **"on the fly"** during run time.
- dynamically allocated space usually placed in a program segment known as the **heap** or the **free store.**

❖ Malloc, calloc & realloc :

- **malloc** is a C standard library function that finds a chunk of free memory of the desired size and returns a pointer to it.
- **Demerits**: 1. May forgot to initialize. 2. Accidentally do something to object before initialization.

- The **calloc** takes two integer arguments. These are multiplied together to specify how much memory to allocate.
- **calloc** initializes all the allocated memory space to **zero** whereas malloc() leaves whatever values may already be there.

e.g. `int *ptrc = (int*)calloc(10, sizeof(int));`

- The realloc Changes the size of the memory block pointed to by ptr.
  The function may move the memory block to a new location

Def: `void* realloc (void* ptr, size_t size);`

e.g : `more_numbers = (int*) realloc (numbers, count * sizeof(int));`

- **Free**: A block of memory previously allocated by a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.

Memory management overhead:

- When malloc request block of memory pool.
- Pool is searched for block of memory large enough to satisfy request.
- This is done by checking map or directory of some sort that shows which block currently in use & which are not.
- It may take several tries before pointer to block is returned.
- Before it returns the size & location of block is recorded so further calls wont use it & when free is called system knows how much memory to be released.

C approach to heap:

```
class Obj
{
    int i, j, k;
    char buf[10];
public:
    void initialize()
    {
        i = j = k = 0;
        memset(buf, 0, 10);
        cout << "Obj initialized";
    }
    void destroy()
    {
        cout << "Obj destroyed";
    }
};
```

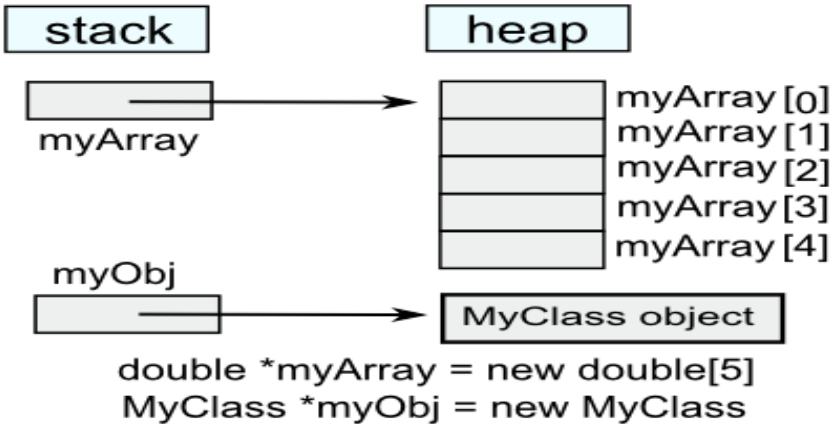
```
int main()
{
    Obj* obj = (obj*)malloc(sizeof(obj));
    obj->initialize();

    obj->destroy();
    free(obj);
}
```

- ## New & Delete

- **new** to create an object dynamically, two things happen as we discussed in the previous section: First, memory is allocated by calling operator new. Second, one or more constructors are called for that memory.
- one or more destructors are called for the memory, and then the memory is deallocated using operator **delete**. Delete called only for object created using new.

e.g :

```
string* pString = new string;
string* pStringArray = new string[10];
delete pString;
delete[] pStringArray;
```



- Delete void pointer is bug.

```
void * ptr = new Obj(10);
delete ptr;
```

If we convert Obj into void compiler will not get type to be call destructor. So we might have memory leak.

Static Allocation	Dynamic Allocation
Memory allocation at compile time.	Memory allocation at run time.
Don't know runtime requirement so may be shortage or wastage of memory.	No shortage or wastage of memory.
Memory free automatically.	Programmer's responsibility to free memory.
Faster	Slower

Casting in C++

- ## static_cast

static_cast does ***compile-time***, not run-time ***checking of the types involved***. In many situations, this can make it the safest type of cast, as it provides the least room for accidental/unsafe conversions between various types.

static_cast is used for cases where you basically want to reverse an implicit conversion, with a few restrictions and additions. static_cast performs no runtime checks. This should be used if you know that you refer to an object of a specific type, and thus a check would be unnecessary.

```
void func(void *data) {
  // Conversion from MyClass* -> void* is implicit
  MyClass *c = static_cast<MyClass*>(data);
  ...
}

int main() {
  MyClass c;
  start_thread(&func, &c)  // func(&c) will be called
      .join();
}
```

In this example, you know that you passed a MyClass object, and thus there isn't any need for a runtime check to ensure this.

e.g. 1

```
int main()
{
    char c;          // 1 Byte
    int* p = &c;     // 4 Bytes
    *p = 5;                 // Pass at compile time, May fail at
    // runtime or corrupt other data at runtime.

    int* ip = static_cast<int*>(&c); //Fail at compile type.Both the
     //pointers are not same type.
}
```

e.g. 2

```
class Base {}
class Derived : private Base {}; // Note private inheritance here.
int main
{
    Derived d1;
    Base* bp1 = (Base*)&d1; //Allowed at compile time but dangourous
    Base* bp2 = static_cast<Base*>(&d1); // Not allowed, Compile error
}


class Base {}
class Derived : public Base {}; // Now public inheritance here.
int main
{
    Derived d1;
```

```
    Base* bp1 = (Base*)&d1; //This is fine but should not used.
    Base* bp2 = static_cast<Base*>(&d1); // This too fine.No Error now.
}
```

Use it for upcast but no for downcast

Consider following example,

```
class Base {}; // We need virtual method here.
class Derived1 : public Base {};
class Derived2 : public Base {};
int main
{
   Derived1 d1;
   Derived2 d2;

   Base* bp1 = static_cast<Base*> (&d1); // Perfectly fine. No issues here
   Base* bp2 = static_cast<Base*> (&d2); // We can convert der to base ptr.

   Derived2* bp2 = static_cast<Derived1*> (&d1); // Here we are converting obj type d1 to d2
   Derived1* bp1 = static_cast<Derived1*> (&d2); // Wrong... We need dynamic cast here..
}
```

If we are sure about the type of pointer then only we should do for downcasting.Because there are no runtime checks are performed on the type. Best way to use dyamic cast here.

- ## dynamic_cast

dynamic_cast exists to facilitate ***run-time checking/conversion*** from a base class to one of its derived classes, via references or pointers to both. If the instance cannot be cast to a derived type, ***dynamic_cast*** will raise a runtime exception ***std::bad_cast*** when attempted on ***references***, or return a ***nullptr*** when applied to ***pointers***.

Due to its purpose being casting down an inheritance hierarchy, dynamic_cast works only when the source type is polymorphic. Otherwise, the compiler will give an error.
dynamic_cast is used for cases where you don't know what the dynamic type of the object is.

```
   if(JumpStm *j = dynamic_cast<JumpStm*>(&stm)) {
      ...
    } else if(ExprStm *e = dynamic_cast<ExprStm*>(&stm)) {
      ...
   }
```

Examples of static_cast and/vs dynamic_cast

If we have the following classes

```
class B {
public:
virtual void func() { };
};

class D : public B {};
```

then you can do the following

```
B* b = new D();
```

```
D* d1 = static_cast<D*> ( b );
// Valid! d1 is a valid and correct pointer to a D
D* d2 = dynamic_cast<D*> ( b );
// Valid! d2 is a valid and correct pointer to a D
```

In this example, both pointers d1 and d2 will point to a correctly typed version of b as requested.
A potential for mishap where static_cast can be more unsafe than dynamic_cast is illustrated by the following example:

```
B* b = new B();
D* d1 = static_cast<D*>(b); // works, but D component is invalid!
D* d2 = dynamic_cast<D*>(b); // cast fails => d2 is now a nullptr
```

Now d1 will point to a data segment of type D*, but the actual data is B*, so attempting to access D-specific member is undefined behaviour.

Dynamic Casts with References

The **dynamic_cast** operator can be used to cast to reference types. C++ reference casts are similar to pointer casts: they can be used to cast *from* references to base class objects *to* references to derived class objects.

e.g.1

```
void Payroll::calc(employee& re) {
    // employee salary calculation
    try {
        manager& rm = dynamic_cast<manager&>(re);
        // use manager::bonus()
    }
    catch (bad_cast) {
        // use employee's member functions
    }
}
```

2.

```
class B
{
public:
    virtual void f1() {};
};

class D :public B
{

};
int main()
{
    D d;
    B b;
    try
    {
        D& ref1 = dynamic_cast<D&>(d); // Cast works
        D& ref2 = dynamic_cast<D&>(b); // Bad cast

    }
    catch (std::bad_cast)
    {
        cout << "Bad cast";
    }
}
```

BOTTOM LINE:
1. Work only on ***polymorphic base class*** (at least one virtual function in base class) because it uses this information to decide about wrong down-cast.
2. It is used for up-cast (D to B) and ***down-cast*** (B to D), but it is mainly used for correct downcast.
3. Using this cast has ***run time overhead***, because it checks object types at run time using ***RTTI*** (Run Time Type Information).
4. If we are sure that we will ***never cast to wrong object then we should always avoid this cast*** and use static_cast.

- ## const_cast

  const_cast is used to cast away the **constness** of variables. Following are some interesting facts about const_cast.

- const_cast can be used to change non-const class members inside a const member function. Consider the following code snippet.
```
// A const function that changes roll with the help of const_cast
void fun() const
{
    (const_cast<student*> (this))->roll = 5;
}
```
- const_cast can be used to pass const data to a function that doesn't receive const.
```
const int val = 10;
const int* ptr = &val;
int* ptr1 = const_cast<int*>(ptr);
cout << fun(ptr1);
```
- const_cast is considered safer than simple type casting. It'safer in the sense that the casting won't happen if the type of cast is not same as original object. For example, the following program fails in compilation because 'int *' is being typecasted to 'char *'
```
int a1 = 40;
const int* b1 = &a1;
char* c1 = const_cast<char*> (b1); // compiler error
```
- const_cast can also be used to cast away volatile attribute. For example, in the following program, the typeid of b1 is PVKi (pointer to a volatile and constant integer) and typeid of c1 is Pi (Pointer to integer)
```
int a1 = 40;
const volatile int* b1 = &a1;
int* c1 = const_cast<int*> (b1);
```
- Ideal use of const_cast is to change non-const variables const pointer. We should not use it to change const pointer of const variable.
```
int val1 = 10; const int val2 = 20;
const int* ptr1 = &val1; const int* ptr2 = &val2;
int* c1 = const_cast<int*> (ptr1); // Ideal use
int* c2 = const_cast<int*> (ptr1); // No error but undefined behavior.
```

BOTTOM LINE:
1. Use const_cast only when you have to.
2. Use const_cast only when the actual refereed object/variable is not const.
3. Use when we are dealing with 3'rd party library and some API want data in non-const form but we have it in const. (actually we can not do anything in that case, but make sure that API is not changing our variable value)

❖ Difference between const & const volatile

- Basically, const means that the value isn't modifiable by the program.
- And volatile means that the value is subject to sudden change (possibly from outside the program).
- In fact, the C Standard gives an example of a valid declaration which is both const and volatile. The example is:
- extern const volatile int real_time_clock;
- where real_time_clock may be **modifiable by hardware**, but cannot be assigned to, incremented, or

decremented.

- In simple terms, Value in 'const volatile' variable **cannot be modified programmatically** but can be modified by hardware. Volatile here is to prevent any compiler optimisation.
- **'const volatile'** we are telling the program not to change it and the compiler that this variable can be changed unexpectedly from input coming from the outside world.

reinterpret_cast In C++ | Where To Use reinterpret_cast In C++? - YouTube
const_cast In C++ | Where To Use const_cast In C++? - YouTube

Special

• Sizeof operator

- Queries size of the object or type.
- Returns size of object or variable.
- Its a operator not a function.
- If applied with type it should use parenthesis.

Why is the size of an empty class not zero in C++?

```
classEmpty {};

int main()
{
      cout <<sizeof(Empty);
      return0;
}
O/P: 1
```

Size of an empty class is not zero. It is 1 byte generally. It is nonzero to ensure that the two different objects will have different addresses.

Size of class with static variable?

```
class A
{
    int a;
    static int b;
};
```

Size of the above class is **4**.

Static variable is not part of the object of a class. There is only one copy of static data shared y all the objects.

They are like global variables accessed through S::b;

• Pre-increament

Pre-increament can be used as l-value where as post-increament does not.

```
int main()
{
    int a = 10;
    ++a = 20;  //Works fine a =11 & then a = 20;
```

```
    cout << a; //20
    a++ = 20;  //Compilation error.
    cout << a;
    return 0
}
```

- ## Tuples in C++

A tuple is **Mar**. The elements can be of different data types. The elements of tuples are initialized as arguments in order in which they will be accessed.
Functions:
1. get() :- get() is used to access the tuple values and modify them, it accepts the index and tuple name as arguments to access a particular tuple element.
2. make_tuple() :- make_tuple() is used to assign tuple with values. The values passed should be in order with the values declared in tuple.

e.g. 1

```
int main()
{
    // Declaring tuple
    tuple <char, int, float, int> geek;

    // Assigning values to tuple using make_tuple()
    geek = make_tuple('a', 10, 15.5, 54);

    // Printing initial tuple values using get()
    cout << "The initial values of tuple are : ";
    cout << get<0>(geek) << " " << get<1>(geek);
    cout << " " << get<2>(geek) << " "<< get<3>(geek) <<endl;

    // Use of get() to change values of tuple
    get<0>(geek) = 'b';
    get<2>(geek) = 20.5;

    char& ch = get<0>(geek);
    ch = 'c'; // This also changes tuple

    cout<< get<4>(geek); // Compilation error out of range

    int i = 1;
    cout<< get<i>(geek); // Compilation error need copile time constant



    // Printing modified tuple values
    cout << "The modified values of tuple are : ";
    cout << get<0>(geek) << " " << get<1>(geek);
    cout << " " << get<2>(geek) << " "<< get<3>(geek) <<endl;


    tuple <char, int, string> t1;
    char x;
    int y;
    string z;
    make_tuple(ref(x), ref(y), ref(z)) = t1;
    std::tie(x,y,z)  = t1; // Did exactly same as above line
    std::tie(x,std::ignore,z)  = t1; // int value can be ignored
```

```
    return 0;
}
```

- ## C++ bitset and its application

A bitset is an array of bool but each Boolean value is not stored separately instead bitset optimizes the space such that each bool takes 1 bit space only, so ***space taken by bitset bs is less than that of bool bs[N] and vector bs(N)***.
However, a limitation of bitset is, ***N must be known at compile time, i.e., a constant*** (this limitation is not there with vector and dynamic array)

```
#define M 32

int main()
{
    // default constructor initializes with all bits 0
    bitset<M> bset1;

    // bset2 is initialized with bits of 20
    bitset<M> bset2(20);

    // bset3 is initialized with bits of specified binary string
    bitset<M> bset3(string("1100"));

    // cout prints exact bits representation of bitset
    cout << bset1 << endl; // 00000000000000000000000000000000
    cout << bset2 << endl; // 00000000000000000000000000010100
    cout << bset3 << endl; // 00000000000000000000000000001100
    cout << endl;

    // declaring set8 with capacity of 8 bits

    bitset<8> set8; // 00000000

    // setting first bit (or 6th index)
    set8[1] = 1; // 00000010
    set8[4] = set8[1]; // 00010010
    cout << set8 << endl;

    // count function returns number of set bits in bitset
    int numberof1 = set8.count();

    // size function returns total number of bits in bitset so there difference
    // will give us number of unset(0) bits in bitset
    int numberof0 = set8.size() - numberof1;

    cout << set8 << " has " << numberof1 << " ones and "
        << numberof0 << " zeros\n";

    // test function return 1 if bit is set else returns 0
    cout << "bool representation of " << set8 << " : ";
    for (int i = 0; i < set8.size(); i++)
        cout << set8.test(i) << " ";

    cout << endl;

    // any function returns true, if atleast 1 bit
    // is set
    if (!set8.any())
```

```
        cout << "set8 has no bit set.\n";

    if (!bset1.any())
        cout << "bset1 has no bit set.\n";

    // none function returns true, if none of the bit
    // is set
    if (!bset1.none())
        cout << "bset1 has some bit set\n";

    // bset.set() sets all bits
    cout << set8.set() << endl;

    // bset.set(pos, b) makes bset[pos] = b
    cout << set8.set(4, 0) << endl;

    // bset.set(pos) makes bset[pos] = 1  i.e. default
    // is 1
    cout << set8.set(4) << endl;

    // reset function makes all bits 0
    cout << set8.reset(2) << endl;
    cout << set8.reset() << endl;

    // flip function flips all bits i.e.  1 <-> 0
    // and  0 <-> 1
    cout << set8.flip(2) << endl;
    cout << set8.flip() << endl;

    // Converting decimal number to binary by using bitset
    int num = 100;
    cout << "\nDecimal number: " << num
        << "  Binary equivalent: " << bitset<8>(num);

    return 0;
}
```

Special 2

❖ Friend class and function in C++

Friend Class

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

```
class Node
{
private:
  int key;
  Node *next;
  /* Other members of Node Class */

  friend class LinkedList; // Now class  LinkedList can
                           // access private members of Node

      Or
  friend int LinkedList::search(); // Only search() of linkedList
                                   // can access internal members


};
```

- Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
- ***Friendship is not mutual***. If a class A is friend of B, then B doesn't become friend of A automatically.
- ***Friendship is not inherited***.
- A friend class declaration cannot define a new class (friend class X {};is an error)

Friend Function

Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:

a) A method of another class

b) A global function

- Being non- member a friend function **can be declared under any access specifier**.
- Friends are non-members hence do not get **"this"** pointer.
- We can make **int main()** as friend of any class.
- We can make **member functions of a class as a friend** of other class.
  friend void B::fB(A& a); //specifying B class member function fB as a friend of A

Why do we need friend functions?

- A friend function can be friendly to 2 or more classes. The friend function does not belong to any class, so it can be used to access private data of two or more classes as in the following example.

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

class Square;  // forward declaration
```

```
class Rectangle {
	int width, height;

public:
	Rectangle(int w = 1, int h = 1):width(w),height(h){}
	friend void display(Rectangle &, Square &);
};

class Square {
	int side;

public:
	Square(int s = 1):side(s){}
	friend void display(Rectangle &, Square &);
};

void display(Rectangle &r, Square &s) {
	cout << "Rectangle: " << r.width * r.height << endl;
	cout << "Square: " << s.side * s.side << endl;
}

int main () {
	Rectangle rec(5,10);
	Square sq(5);
	display(rec,sq);
	return 0;
}
```

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them.

- We cant make member function of same class as a friend function

```
class B
{
public:
	friend void fB() {}; // This will treated as fried function not member of the class B.
};
```

- Both function template and class template declarations may appear with the friend specifier in any non-local class or class template.

```
class A {
    template<typename T>
    friend class B; // every B<T> is a friend of A

    template<typename T>
    friend void f(T) {} // every f<T> is a friend of A
   };
```

❖ Const & Pointer

- You can assign the address of non const object to a const pointer because you are simply promising not

to change something that is ok to change.

```
int m;
int * const n = &m;//Ok
```

- You cant assign the address of a const object to a non const pointer because then you are simply saying you might change the object via the pointer.

```
const int m = 2;
int * n = &m;//Illegal
```

❖ Passing Const by value

- Original value of the variable will not be changed by the const function.

```
void func (const int i)
{
      i++; // Illegal
}
```

❖ typedef

- Used for name aliasing.
- Used when data type gets slightly complicated.

e.g.

```
typedef unsigned long ulong;
typedef int* IntPtr;
IntPtr x,y;
typedef struct{
char c;
int i;
}structorone;
```

Virtual Functions and Runtime Polymorphism in C++

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform ***Late Binding*** on this function.

virtual Keyword is used to make a member function of the base class Virtual.

e.g.

```
class Base
{
 public:
 virtual void show()
```

```
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}

int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Late Binding Ocuurs
}
```

O/P : Derived class

- We can call private function of derived class from the base class pointer with the help of virtual keyword. In above example if we change code like this,

```
class Derived
{
private:
    virtual void show()
    {
        cout << "Derived class\n";
    }
};
```

Still O/P : Derived class

(Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.)

- If declared virtual in base class it becomes virtual in all derived class.
- Compiler creates ***single vtable*** for each class which contains virtual function & its addresses.
- Class contains virtual function or if base class contains virtual function compiler will create unique vtable for that class.
- If you dont override virtual function in derived class then in vtable address for the virtual function is taken from its base class.
- *vptr* points to starting address of appropriate **VTABLE**.
- *vptr* is properly initialized by compiler in Constructor.
- If constructor is not present it will be initialized in default constructor.

❖ this pointer

this pointer is passed as hidden argument to all non static function calls & is available as local variable within functions. It is a constant pointer which holds the memory address of current object.

For class X the type of this pointer is

X* **const**

this is an rvalue (you cannot take its address).

Since this itself is not a pointer that you can modify or deem to really be "stored" anywhere — recall that its value is automatically determined by the compiler based on context — the committee decided that it would be best off as an rvalue expression.

this, as a pointer, certainly contains a memory address as its physical value (on typical systems) but that doesn't mean it has one for itself. Try taking its address: &this can't work.

e.g:

```
class MyClass
{
public:
    void fun()
    {
        MyClass* ptr = &this; // Error: & requires l-value;
    }
};
```

Depending on the compiler and the target architecture, it will often stored in a register or stack or heap.

Following are the situations where 'this' pointer is used:

1. **When local variable's name is same as member's name**

```
class Test
{
private:
   int x;
public:
    void setX(int x)    {
           // The 'this' pointer is used to retrieve the object's x
           // hidden by the local variable 'x'
           this->x = x;
    }
}
```

2. **To return reference to the calling object**

```
/* Reference to the calling object can be returned */
Test& Test::func()
{
    // Some processing
    return *this;
}
```

Deleting this pointer

Ideally we should not delete 'this' pointer.

However if used then works only for objects allocated using operator new.

```
class A
{
    int x;
public:
```

```
    void fun()
    {
        delete this;
    }
};
int main()
{
    A* ptr = new A;
    ptr->fun(); //Works well & delete this
    ptr = NULL;

    A a1;
    a1.fun();   //Runtime error should not use with stack variable.
    return 0;
}
```

Once delete this is done, any member of the deleted object should not be accessed.

```
void fun()
{
    delete this;
    cout << x; //undefined behaviour.
}
```

❖ Abstract class

- Defines behavior or capabilities of class without committing to a particular implementation of the class.
- Class is made abstract by declaring at least one of its functions as pure virtual function.

e.g.

```
class Box{
private:
double length;
public:

    virtual void show() = 0;
};
```

- Purpose is to provide an appropriate base class from which other classes can inherit.
- Cannot be used to instantiate & attempting to do so give compilation error.
- When abstract class is inherited all pure virtual functions must be implemented or inherited class becomes abstract as well.

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

- This will create slot for a function in vtable but not to put address in that particular slot. So even one pure virtual function in class made VTABLE incomplete.
- So when someone tries to create object of that class compiler will give error.

❖ Virtual destructor

It is usefull when you delete an instance of a derived class through a pointer to base class.

```
class Base
{
    //Some virtual method
};
class Derived : public Base
{
    ~Derived()
    {
        //Cleanup
    }
};
Base* b = new Derived;
//Use b
delete b;
```

Since base destructor is not virtual & b is a Base*

- It will call base destructor only resulting in memory leak. So make base class destructor virtual.
- By doing this compiler call derived class destructor.

❖ Pure Virtual destructor

Used when you want to create an abstract base class

- Can't be instantiated
- Need virtual destructor behavior.
- But does not need any other virtual dispatch.

```
class Base
{
    virtual Base() = 0; //Pure virtual method
};
Base::~Base()
{
}
```

```
    // Yes Pure virtual destructor can have body also.
```

❖ Some points about virtual function

- ***Constrctor can not be virtual function*** because it needs to know the exact type to create
- ***Calling virtual function is generally slower*** than calling a normal method because vtable pointer must be derefernced in order to find the address of function to call.
- In addition each object contains at least one virtual function must have an additional pointer allocated within it. So may impose some memory efficiency.
- Default arguments do notparticipate in signature of function.

```
class Base
{
public:
    virtual void fun(int x = 0) { cout << x; }
};

class Derived : public Base
{
public:
    virtual void fun(int x) { cout << x; }
};
int main()
{
    Derived d1;
    Base* ptr = &d1;
    ptr->fun();
}
```

O/P: 0

Here so signature of base class and derived class are considered same hence function of base class is overridden.

Also default value is used at compile time when compiler sees that argument is missing in a function call it substitutes with default value.

Now consider another code,

```
class Base
{
public:
    virtual void fun(int x = 0) { cout << x; }
};

class Derived : public Base
{
public:
    virtual void fun(int x = 10) { cout << x; }
};
int main()
{
    Derived d1;
    Base* ptr = &d1;
    ptr->fun();
}
O/P : 0
```

By default value is **substituted at compile time**. The fun() is called on ptr which is pointer of base. So compiler substitutes 0 (Not 10.)

Can we call private virtual function:

Yes. Consider following scenario,

```
class Base
{
public:
    virtual void fun(int i) {}
};

class Derived : public Base
{
private:
    virtual void fun(int x) {}
};

1.
int main()
{
} //Works fine...

2.
int main()
{
    Derived d;
    d.fun(); //Compiler error fun is private in Derived class.
    return 0;
}

3.

int main()
{
    Base* ptr = new Derived;
    ptr->fun(10); // Derived function called.
    return 0;
}
```

fun()is public in Base class. Access specifier are checked at copile time & fun() is public in Base class. At runtime only function corresponding to the pointed object is called & access specifier is not checked.
So private function of Derived class is being called through a pointer of Base class.

Virtual function in Constructor:

```
class Base
{
public:
    Base() { fun(); }
    virtual void fun() { cout << "Base"; }
};

class Derived : public Base
{
public:
    virtual void fun(int x) { cout << "Derived"; }
};


int main()
{
    Derived d;
    return 0;
}
O/P: Base
```

Reason is base classes are constructed before derived class. So before Derived, Base must be made. When

Base's constructor called its not a Derived yet, so the virtual function table still has the entry for Base's copy of func() and Derived's vtable yet to be created.

Static Polimorphism/ Simulated Polimorphism

Everyone knows how to use the Polymorphism with dynamic binding. But dynamic binding has a run-time cost of both time and memory (virtual table). Following program demos a polymorphism with static binding, which has the benefits of plymorphism without the cost of virtual table.

```
template<typename T>
class Shape
{
public:
    Shape() {}
    void call() {
        cout << "Shape call" <<endl;
        fun();
    }
    void fun() {
        cout << "Shape function" << endl;;
        static_cast<T*> (this)->fun();
    }
};

class Circle : public Shape<Circle>
{
public:
    Circle() {}
    void fun() {
        cout << "Circle function" << endl;;
    }
};

int main()
{
    Circle c1;
    c1.call();
}
O/P:

Shape call
Shape function
Circle function
```

Initializer list in C++

❖ When do we use Initializer List in C++?

1. For initialization of non-static const data members

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

class Test {
    const int t;
public:
    Test(int t):t(t) {}  //Initializer list must be used
    int getT() { return t; }
};

int main() {
    Test t1(10);
    cout<<t1.getT();
    return 0;
}

/* OUTPUT: 10 */
```

2. For initialization of reference members

Reference members must be initialized using Initializer List.

```
class Test {
     int &t;
public:
    Test(int &t):t(t) {}  //Initializer list must be used
  int getT() { return t; }
};
```

3. For initialization of member objects which do not have default constructor

In the following example, an object “a” of class “A” is data member of class “B”, and “A” doesn’t have default constructor. Initializer List must be used to initialize “a”.

```
class A {
    int i;
public:
    A(int arg)
    {
        i = arg;
    }
};

// Class B contains object of A
class B {
    A a;
public:
```

```
        B(int x):a(x) {  //Initializer list must be used
        cout << "B's Constructor called";
    }

    };
```

If class A had both default and parameterized constructors, then Initializer List is not must

4. For initialization of base class members

Like point 3, parameterized constructor of base class can only be called using Initializer List.

```
    class A {
        int i;
    public:
        A(int arg) {
            i = arg;
            cout << "A's Constructor called: Value of i: " << i;
        }
    };

    // Class B is derived from A
    class B: A {
    public:
        B(int x):A(x) { //Initializer list must be used
            cout << "B's Constructor called";
        }
};
```

5. When constructor's parameter name is same as data member

```
   class A {
        int i;
   public:
        A(int i):i(i) { }  // Either Initializer list or this pointer
                           must be used
    /* The above constructor can also be written as
       A(int i) {
       this->i = i;
         } */
       };
```

6. For Performance reasons

```
class MyClass {
    Type variable;
public:
    MyClass(Type a) {  // Assume that Type is an already
                       // declared class and it has appropriate
                       // constructors and operators
      variable = a;
    }
};
```

Here compiler follows following steps to create an object of type MyClass

1. Type's constructor is called first for "a".
2. The assignment operator of "Type" is called inside body of MyClass() constructor to assign

```
variable = a;
```

3. And then finally destructor of “Type” is called for “a” since it goes out of scope.

Now consider the same code with MyClass() constructor with Initializer List

```
// With Initializer List
class MyClass {
    Type variable;
public:
    MyClass(Type a):variable(a) {   // Assume that Type is an already
                    // declared class and it has appropriate
                    // constructors and operators
    }
};
```

With the Initializer List, following steps are followed by compiler:
1. Copy constructor of “Type” class is called to initialize : variable(a). The arguments in initializer list are used to copy construct “variable” directly.
2. Destructor of “Type” is called for “a” since it goes out of scope.

Here we can see we will skip assignment operator call which will be considerable if object has n number of member variable.

❖ Preventing pass by value

- To prevent pass by value of object declare a private copy constructor, you even don’t need to create a definition unless one of your member function or a friend function needs to perform a pass by value.
- If user pass object by value compiler will give error “copy constructor is private”. Compiler will not create a default copy constructor as you explicitly stated that.

```
class NoCC
{
private:
      int i;
      NoCC(const NoCC& val)
      {
            i = val.i;
      }
public:
      NoCC(int val=0):i(val)
      {
      }

};

void func(NoCC obj)
{
}
int main()
{
      NoCC obj1;
      func(obj1);           //Compilation error: Calls copy constructor
      NoCC obj2 = obj1;     //Compilation error: Calls copy constructor
```

```
    NoCC obj3 (obj1);     //Compilation error: Calls copy constructor
}
```

Move semantics and r-value reference

❖ lvalues references and rvalues references in C++ with Examples

"l-value" refers to a memory location that identifies an object. **"r-value"** refers to the data value that is stored at some address in memory. **References in C++** are nothing but the alternative to the already existing variable. They are declared using the **'&'** before the name of the variable.

Rvalue references is a small technical extension to the C++ language. Rvalue references allow programmers to **avoid logically unnecessary copying** and to provide perfect forwarding functions. They are primarily meant to aid in the design of higer performance and more robust libraries.

```
int a = 10;
// Declaring lvalue reference
int& lref = a;

// Declaring rvalue reference
int&& rref = 20;
```

- rvalue references have two properties that are useful:

- rvalue references extend the lifespan of the temporary object to which they are assigned.
- Non-const rvalue references allow you to modify the rvalue.

- An rvalue reference behaves just like an lvalue reference except that it can bind to a temporary (an rvalue), whereas you can not bind a (non const) lvalue reference to an rvalue.

```
A&  a_ref3 = A();  // Error!
A&& a_ref4 = A();  // Ok
```

Important: lvalue references can be assigned with the **rvalues** but **rvalue** references cannot be assigned to the **lvalue**.

```
int&& ref1 = 20;  // Ok
int& ref2 = ref1; // Ok

But
    int var = 10;
    int& ref1 = var;// OK
    int& ref1 = 10;// Error
    const int& ref1 = 10;// OK. Compiler internally does some trick here.
    int&& ref2 = var;// Error : You cannot bind an lvalue to an rvalue reference
    int&& ref2 = ref1;// Error : You cannot bind an lvalue ref to an rvalue reference
```

- Uses of rvalue references:

- They are used in working with the move constructor and move assignment.
- cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'.

- cannot bind rvalue references of type ‘int&&‘ to lvalue of type ‘int’.

```
void printReferenceValue(int&& x)
{
    cout << x << endl;
}


int main()
{

    int var = 10;
    printReferenceValue(var); //  Error: cannot convert argument 1 from 'int' to 'int &&'
                              // You cannot bind an lvalue to an rvalue reference
    printReferenceValue(100); // Works fine.
}
```

❖ Eliminating spurious copies

Copying can be expensive. For example, for std::vectors, v2=v1 typically involves a function call, a memory allocation, and a loop. This is of course acceptable where we actually need two copies of a vector, but in many cases, we don't: We often copy a vector from one place to another, just to proceed to overwrite the old copy.

Consider:

```
template <class T> swap(T& a, T& b)
{
    T tmp(a);   // now we have two copies of a
    a = b;      // now we have two copies of b
    b = tmp;    // now we have two copies of tmp (aka a)
}
```

But, we didn't want to have any copies of a or b, we just wanted to swap them. Let's try again:

```
template <class T> swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

This move() gives its target the value of its argument, but is not obliged to preserve the value of its source.

```
template <class T>
class clone_ptr
{
private:
    T* ptr;
public:
    // construction
    explicit clone_ptr(T* p = 0) : ptr(p) {}

    // destruction
    ~clone_ptr() { delete ptr; }

    // copy semantics
    clone_ptr(const clone_ptr& p)
        : ptr(p.ptr ? p.ptr->clone() : 0) {}
    // move semantics
    clone_ptr(clone_ptr&& p)
        : ptr(p.ptr) {
        p.ptr = 0;
    }
};
```

```
clone_ptr<base> p1(new derived);
 // ...
clone_ptr<base> p2 = p1;  // p2 and p1 each own their own pointer
```

But

```
clone_ptr<base> p1(new derived);
 // ...
clone_ptr<base> p2 = std::move(p1);  // p2 now owns the pointer intead of p1
```

❖ Move semantics & move constructor

- **Limitations of reference:**

  We can create referece of variable, but we **can not create a reference of temp variable**. Even though we can create const reference of temporary variable, we can not modify it.

  ```
  int a = 10;
  int& ref1 = a; // OK
  int& ref2 = 20; // Not ok
  const int& ref2 = 20; // Ok
  ref2 = 30; // But this is not
  ```

  So solution to this is **rValue reference**.

- **rValue referece**

  int && rValRef = 20; // Works fine

  rValRef = 15; // We can modify too.

On declaring the new object and assigning it with **the r-value**, firstly a temporary object is created, and then that temporary object is used to assign the values to the object. Due to this the **copy constructor** is called several times and increases the overhead and decreases the computational power of the code. To ***avoid this overhead and make the code more efficient*** we use move constructors.

- **Why Move Constructors are used?**

Move constructor moves the resources in the heap, i.e., unlike copy constructors which copy the data of the existing object and assigning it to the new object move constructor just makes the pointer of the declared object to point to the data of temporary object and nulls out the pointer of the temporary objects. Thus, move constructor prevents unnecessarily copying data in the memory.

Check following program,

```
class Test
{
private:
    int* ptr;
public:
    Test()
        {
            cout << " Const" << endl;
            ptr = new int(0);
        }
    Test(const Test & val)
        {
            cout << " Copy Const" << endl;
            ptr = new int;
```

```
            *ptr = *val.ptr;
        }
    ~Test()
        {
            cout << " Destr" << endl;
            delete ptr;
        }
};

int main()
{
    std::vector<Test> testList;
    testList.push_back(Test());
    return 0;
}
```

O/P:

Const

Copy cost

Destr

Destr

Here we are assigning memory two times, first time in Costructor & in copy constructor this increases the overhead and decreases the computational power. So we can write move constructor for the class like,

```
    Test(Test&& val)
    {
        cout << " Move Const" << endl;
        ptr = val.ptr;
        val.ptr = nullptr;
    }
```

O/P:

```
 Const
 Move Const
 Destr
 Destr
```

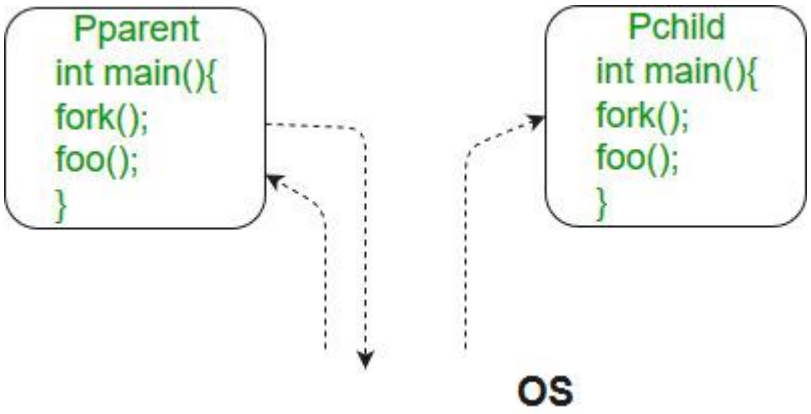
- Work of move constructor looks a bit like default member-wise copy constructor but in this case, it ***nulls out*** the pointer of the temporary object preventing more than one object to point to same memory location.
- The ***unnecessary call to the copy constructor is avoided*** by making the call to the ***move constructor***. Thus making the code more memory efficient and decreasing the overhead of calling the move constructor.
- Now we can explicitly call move constructor for normal variables too.

```
Test t1;
std::vector<Test> testList;
testList.push_back(std::move(t1));
```

❖ fork() in C

- Fork system call is used for ***creating a new process***, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process)
- After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
- It takes no parameters and returns an integer value. Below are different values returned by fork()

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.



e.g:

1.

```
int main()
{

    // make two process which run same program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output:

```
Hello world!
Hello world!
```

2.

```
int main()
{
```

```
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Output:

hello

hello

hello

hello

hello

hello

hello

hello

3.

```
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

O/P:

Hello from Child!

Hello from Parent!

(or)

2.

Hello from Parent!

Hello from Child!

Rvalue References and Perfect Forwarding in C++0x

The key difference is of course that an rvalue reference can bind to an rvalue, whereas a non-const lvalue reference cannot. This is primarily used to support *move* semantics for expensive-to-copy objects:

❖ Perfect Forwarding

When you combine rvalue references with function templates you get an interesting interaction: if the type of a function parameter is an rvalue reference to a template type parameter then the type parameter is deduce to be an lvalue reference if an lvalue is passed, and a plain type otherwise. This sounds complicated, so lets look at an example:

```
template<typename T>
void f(T&& t);

int main()
{
    X x;
    f(x);   // 1
    f(X()); // 2
}
```

The function template f meets our criterion above, so in the call f(x) at the line marked "1", the template parameter T is deduced to be X&, whereas in the line marked "2", the supplied parameter is an rvalue (because it's a temporary), so T is deduced to be X.

Why is this useful? Well, it means that a function template can pass its arguments through to another function whilst ***retaining the lvalue/rvalue nature of the function arguments*** by using std::forward. This is called "***perfect forwarding***", ***avoids excessive copying, and avoids the template author having to write multiple overloads for lvalue and rvalue references. Let's look at an example:***

```
void g(X&& t); // A
void g(X& t);      // B

template<typename T>
void f(T&& t)
{
    g(std::forward<T>(t));
}

void h(X&& t)
{
    g(t);
}

int main()
{
    X x;
    f(x);   // 1
    f(X()); // 2
    h(x);
    h(X()); // 3
}
```

This time our function f forwards its argument to a function g which is overloaded for lvalue and rvalue references to an X object. g will therefore accept lvalues and rvalues alike, but overload resolution will bind to a different function in each case.

At line "1", we pass a named X object to f, so T is deduced to be an lvalue reference: X&, as we saw above. When T is an lvalue reference, std::forward<T> is a no-op: it just returns its argument. We therefore call the overload of g that takes an lvalue reference (line B).

At line "2", we pass a temporary to f, so T is just plain X. In this case, std::forward<T>(t) is equivalent to static_cast<T&&>(t): it ensures that the argument is forwarded as an rvalue reference. This means that the overload of g that takes an rvalue reference is selected (line A).

This is called ***perfect forwarding*** because the same overload of g is selected as if the same argument was supplied to g directly. It is essential for library features such as std::function and std::thread which pass arguments to another (user supplied) function.

Note that this is unique to template functions: we can't do this with a non-template function such as h, since we don't know whether the supplied argument is an lvalue or an rvalue. Within a function that takes its arguments as rvalue references, the named parameter is treated as an lvalue reference. Consequently the call to g(t) from h always calls the lvalue overload. If we changed the call to g(std::forward<X>(t)) then it would always call the rvalue-reference overload. The only way to do this with "normal" functions is to create two overloads: one for lvalues and one for rvalues.

Now imagine that we remove the overload of g for rvalue references (delete line A). Calling f with an rvalue (line 2) will now fail to compile because you can't call g with an rvalue. On the other hand, our call to h with an rvalue (line 3) will still compile however, since it always calls the lvalue-reference overload of g. This can lead to interesting problems if g stores the reference for later use.

• Type Inference in C++ (auto and decltype)

Before C++ 11, each data type needs to be explicitly declared at compile time, limiting the values of an expression at runtime but after the new version of C++, many keywords are included which allows a programmer to leave the type deduction to the compiler itself.

With type inference capabilities, we can spend less time having to write out things compiler already knows. As all the types are deduced in compiler phase only, the time for compilation increases slightly but it does not affect the run time of the program.

❖ auto keyword

The auto keyword specifies that the type of the variable that is being declared will be **automatically deducted from its initializer**. In case of functions, if their return type is auto then that will be evaluated by return type expression at runtime.

```
// auto a; this line will give error because 'a' is not initialized at
 // the time of declaration a=33;

 // see here x ,y,ptr are initialised at the time of
 // declaration hence there is no error in them
 auto x = 4;
 auto y = 3.37;
 auto ptr = &x;
 cout << typeid(x).name() << endl
      << typeid(y).name() << endl
      << typeid(ptr).name() << endl;
```

- **Robustness:** If the expression's type is changed—this includes when a function return type is changed—it just works.
- **Performance:** You're guaranteed that there will be no conversion.
- **Usability:** You don't have to worry about type name spelling difficulties and typos.
- **Efficiency:** Your coding can be more efficient.

❖ References and cv-qualifiers

Note that using **auto** drops **references**, **const** qualifiers, and **volatile** qualifiers.

```
int main( )
{
    int count = 10;
    int& countRef1 = count;
    int& countRef2 = count;

    auto myAuto = countRef1; // myAuto is an int, not an int reference
    decltype(auto) myAuto = countRef1; // Now this is int&

    const int x = 0;
    auto myAuto = x ; // myAuto is an int, not an const int reference
    decltype(auto) myAuto = x; // Now this is const int

    int&& z = 0;
    auto myAuto = z ; // myAuto is an int, not an int&& reference
    decltype(auto) myAuto = z; // Now this is int&&


    countRef = 11;
    cout << count << " ";

    myAuto = 12;
    cout << count << endl;

}

O/P: 11 11

int main()

{
    // std::initializer_list<int>
    auto A = { 1, 2 };

    // std::initializer_list<int>
    auto B = { 3 };

    // int
    auto C{ 4 };

    // C3535: cannot deduce type for 'auto' from initializer list'
    auto D = { 5, 6.7 };

    // C3518 in a direct-list-initialization context the type for 'auto'
    // can only be deduced from a single initializer expression
```

```
    auto E{ 8, 9 };

    return 0;

}
```

❖ decltype Keyword

- It ***inspects the declared type of an entity or the type of an expression***. ***Auto lets you declare a variable with particular type*** whereas ***decltype lets you extract the type from the variable*** so decltype is sort of an operator that evaluates the type of passed expression.
- The **decltype** type specifier, together with the auto keyword, is **useful** primarily to developers who write **template libraries.**

```
1.
// C++ program to demonstrate use of decltype
#include <bits/stdc++.h>
using namespace std;

int fun1() { return 10; }
char fun2() { return 'g'; }

int main()
{
    // Data type of x is same as return type of fun1()
    // and type of y is same as return type of fun2()
    decltype(fun1()) x;
    decltype(fun2()) y;

    cout << typeid(x).name() << endl;
    cout << typeid(y).name() << endl;

    return 0;
}
```

O/P:

i

c

```
2.
template<typename T1, typename T2>
T2 add(T1 a, T2 b) { return a+b;}
int main()
{
      add(1,1.8); // O/P: 2.8
      add(1.8,1); // O/P: 2 because it is of type T2
}
```

To solve this issue we need to

auto add `(T1 a, T2 b)->decltype(a+b){ return a+b;}`

`O/P: 2.8  2.8`

To `decltype` checks expression type here & gives as float.So the o/p of the program in both the cases are in float. This is also called as **trailing return type** syntax.

3.

```
// Another C++ program to demonstrate use of decltype
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int x = 5;

    // j will be of type int : data type of x
    decltype(x) j = x + 5;

    cout << typeid(j).name();

    return 0;
}
```

O/P: i

4.

```
//We can give auto return type like this,
auto f1() ->decltype(10)
{
   return 10;
}
```

decltype vs typeid:

decltye gives the type information at **compile time** while **typeid** gives at **runtime**. So if we have base class reference (or pointer) referring to (or pointing to) a derived class object, the **decltype would give type as base class reference** (or pointer, but **typeid would give the derived type reference** (or pointer).

❖ decltype vs auto

decltype gives the *declared* type of the expression that is passed to it.

auto does the same thing as template type deduction. So, for example, if you have a function that returns a reference, auto will still be a value (you need auto& to get a reference), but decltype will be exactly the type of the return value

```
#include <iostream>
int global{};
int& foo()
{
    return global;
}

int main()
{
    decltype(foo()) a = foo(); //a is an `int&`
    auto b = foo(); //b is an `int`
}
```

You can even use decltype(expr) to specify a base class:

```
auto foo = [](){return 42;};
```

```
class DerivedFunctor : public decltype(foo)
{
  public:
    MyFunctor(): decltype(foo)(foo) {}

  // ...
};
```

❖ auto vrs template

The only thing which makes it the auto keyword different from template that is you cannot make a generic class using the auto keyword.

```
class B { auto a; auto b; }
```

❖ extern c

When you are writing C++ code and including c code in that.

```
#include"cfile.h" // This is header of c file where fun is present.
int main()
{
      fun();  // This will give linker error as fun name was mangled in C++
             //code and it is not in c code where name mangling is absent.
      return;
}

So to avoid that we use extern c which tells compiler not to do name magling
for functions in c file.
extern "C" {#include"cfile.h"}
```

❖ Stop taking address of your object

- Overload & operator and keep it private.
- Delete & operator from your class.

❖ Hiding function name in derived class

```
class Base
{
public:
    int print(int val) {}
};
class Derived : public Base
{
public:
    int print(char val) {}

};

int main()
```

```
{
    Derived d;
    d.print('a');
    d.print(10); // Both calls derived fun.
    d.Base::print(10); // This calls base class fun.
}
Or do
class Derived : public Base
{
       public:
       using Base::fun; // This don't hide all fun functions from base class.
       int print(char val) {}
};
```

If Derived class dosent contain matching parameter for fun then compiler will give error.

```
class Derived : public Base
{
       public:

       int print(char val1, int val2) {} // Error for line d.print(5)
}
```

❖ Understanding constexpr specifier in C++

The main idea is performance improvement of programs by doing computations at compile time rather than run time. Note that once a program is compiled and finalized by developer, it is run multiple times by users. The idea is to spend time in compilation and save time at run time

constexpr specifies that the value of an object or a function can be **evaluated at compile** time and the expression can be used in other constant expressions. For example, in below code product() is evaluated at compile time.

```
// constexpr function for product of two numbers. By specifying constexpr, we suggest
compiler to  to evaluate value at compile time.

constexpr int  product(int x, int y)
{
    return (x * y);
}
int main()
{
    const int x = product(10, 20);
    cout << x;
    return 0;
}
O/P : 200
```

A function be declared as constexpr

1. In C++ 11, a constexpr function should contain only one return statement. C++ 14 allows more than one statements.
2. constexpr function should refer only constant global variables.
3. constexpr function can call only other constexpr function not simple function.
4. Function should not be of void type and some operator like prefix increment (++v) are not allowed in constexpr function.

constexpr vs inline functions:
Both are for performance improvements, inline functions request compiler to expand at compile time and save time of function call overheads. In *inline functions, expressions are always evaluated at run time*. constexpr is different, here *expressions are evaluated at compile time*.

constexpr with constructors:
constexpr can be used in constructors and objects also. See this for all restrictions on constructors that can use constexpr.

```
//  A class with constexpr constructor and function
class Rectangle
{
    int _h, _w;
public:
    // A constexpr constructor
    constexpr Rectangle (int h, int w) : _h(h), _w(w) {}

    constexpr int getArea ()  {   return _h * _w; }
};
```

constexpr vs const
They serve different purposes. constexpr is mainly for optimization while const is for practically const objects like value of Pi.
Both of them can be applied to member methods. Member methods are made const to make sure that there are no accidental changes by the method. On the other hand, the idea of using constexpr is to compute expressions at compile time so that time can be saved when code is run.
const can only be used with non-static member function whereas constexpr can be used with member and non-member functions, even with constructors but with condition that argument and return type must be of literal types.

❖ Nested Classes in C++

- A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.
- For example, program 1 compiles without any error and program 2 fails in compilation.

Program 1

```
#include<iostream>

using namespace std;

 /* start of Enclosing class declaration */
class Enclosing {

   int x;

   /* start of Nested class declaration */
   class Nested {
      int y;
      void NestedFun(Enclosing *e) {
        cout<<e->x;  // works fine: nested class can access
                     // private members of Enclosing class
      }
```

```
   }; // declaration Nested class ends here
}; // declaration Enclosing class ends here

int main()
{

}
```

Run on IDE

Program 2

```
#include<iostream>

using namespace std;

 /* start of Enclosing class declaration */
class Enclosing {

   int x;

   /* start of Nested class declaration */
   class Nested {
      int y;
   }; // declaration Nested class ends here

   void EnclosingFun(Nested *n) {
        cout<<n->y;  // Compiler Error: y is private in Nested
   }
}; // declaration Enclosing class ends here

int main()
{

}
```

❖ Composition

- Composition is a kind of association where the composite object ***has sole responsibility for the disposition of the component parts.***
- The relationship between the composite and the component is a strong ***"has a"*** relationship, as the composite object takes ownership of the component. This means the composite is responsible for the creation and destruction of the component parts.

E.g:

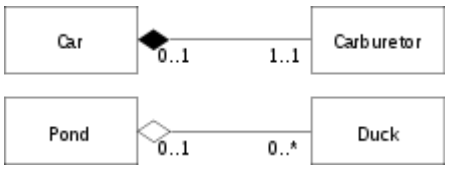
```
// Composition
class Car
{
private:

  // Car is the owner of carburetor.
  // Carburetor is created when Car is created,
  // it is destroyed when Car is destroyed.
  Carburetor carb;

};
```

❖ Aggregation

- Aggregation is a kind of association that specifies a whole/part relationship between the aggregate (whole) and component part.
- This relationship between the aggregate and component is a weak “has a” relationship, as the component may survive the aggregate object.
- The component object may be accessed through other objects without going through the aggregate object.
- The aggregate object does not take part in the lifecycle of the component object, meaning the component object may outlive the aggregate object.

E.g:

```
// Aggregation
class Pond
{
private:

  // Pond is not the owner of ducks,
  // it has references on other ducks managed somewhere else
  std::vector<Duck*> ducks;
};
```



UML notation for a composition (upper) and an aggregation (lower)

❖ Order of constructor & destructor

Suppose we have following example

```
class base {
public:
    base() : {
              cout << "base constructor" << endl;
    }

    ~base() {
        cout << "~base destructor" << endl;
    }
};

class derived1 : public base{
     Member m1, m2;
public:
       derived1(int val) :m2(val),m1(val) {
        cout << "derived1 constructor" << endl;
    }


    ~derived1() {
              cout << "~derived1 destructor" << endl;
       }
};
```

```
class derived2 : public derived1{
    Member m3, m4;
public:
       derived2(int val) :m3(val),m4(val) {
        cout << "derived2 constructor" << endl;
    }


    ~derived2() {
              cout << "~derived2 destuctor" << endl;
       }
};

int main()
{
       derived2 d1(10);
}
```

O/P:

```
base constructor
member1 constructor
member2 constructor
derived1 constructor
member3 constructor
member4 constructor
derived2 constructor
derived2 destructor
member4 destructor
member3 destructor
derived1 destructor
member2 destructor
member1 destructor
base destructor
```

- Order of constructor call for member class is completely unaffected by the order of the calls in the constructor initializer list.

❖ Internal & External linkage

- When you write an implementation file (.cpp, .cxx, etc) your compiler generates a translation unit. This is the object file from your implementation file plus all the headers you #included in it.
- Internal linkage refers to everything only in scope of a translation unit.
- External linkage refers to things that exist beyond a particular translation unit. In other words,accessible through the whole program, which is the combination of all translation units (or object files).
- A global variable has external linkage by default. Its scope can be extended to files other than containing it by giving a matching extern declaration in the other file.
- The scope of a global variable can be restricted to the file containing its declaration by prefixing the declaration with the keyword static. Such variables are said to have internal linkage.

```
void f(int i);
extern const int max = 10;
int n = 0;
static float z = 0.0;

int main(){
```

```
}
```

- The signature of function f declares f as a **function with external linkage**(default). Its definition must be provided later in this file or in other translation unit (given below).
- max is defined as an integer constant. The **default linkage for constants is internal**. So that max can be accessed in other files. Its linkage is made external with the keyword extern.
- n is defined as an integer variable. The default linkage for variables defined outside function bodies is external.
- Static variable: Scope of this is for a file. It is accessible every where in the file in which it is declared. It resides in the DATA segment of RAM. Since this can only be accessed inside a file and hence **INTERNAL** linkage. Any other files cannot see this variable.

❖ Can main() be overloaded in C++?

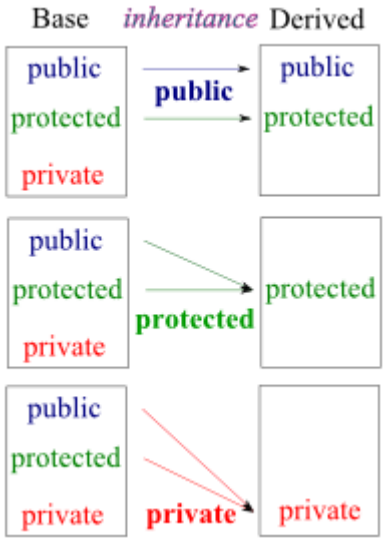
```
int main(int a)
{
}
int main(char* a)
{
}
```

- The above program **fails in compilation** and produces warnings and errors (See this for produced warnings and errors). You may get different errors on different compilers.
- To overload main() function in C++, it is necessary to use class and declare the main as member function. Note that main is not reserved word in programming languages like C, C++, Java and C#.

```
int main()
{
    int main = 10; //Ok
}

class Test
{
public:
    int main(int s)
    {
        return 0;
    }
    int main(char* s) //Ok We can overload main in class.
    {
        return 0;
    }
}


void main() {}
```

- This leads to compilation error on some compiler and warning with some compiler.

Inheritance in C++



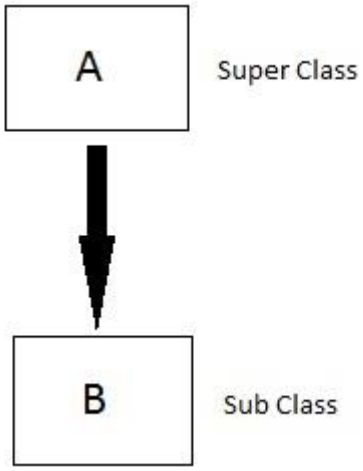
- ## Type of Inheritance

- we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

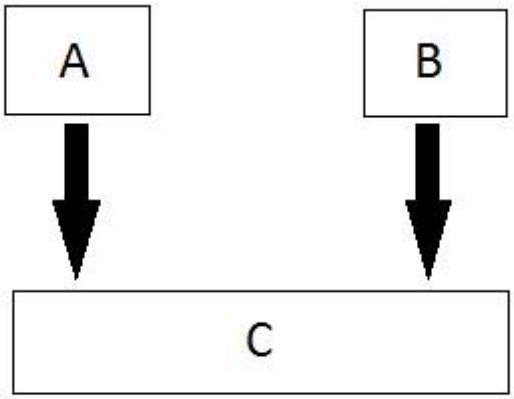
❖ Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



❖ Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



e.g:

```
class Engineer
{
public:
    Engineer() { cout << "Engineer" << endl; }
    void work() { cout << "Engineers work" << endl; }
    void doProgram() { cout << "Programming work" << endl; }
};
class Youtuber
{
public:
    Youtuber() { cout << "Youtuber" << endl; }
    void work() { cout << "Youtubers work" << endl; }
    void doShooting() { cout << "Shooting work" << endl; }
};
class Rupesh : public Engineer, public Youtuber
{
```

```
public:
    Rupesh() { cout << "Rupesh" << endl; }
};

int main()
{
    Rupesh r1;
    return 0;
}

O/P:

Engineer
Youtuber
Rupesh

// Depends on how we derive the class.

If we do
class Rupesh: public Youtuber, public Engineer

Then O/P:
Youtuber
Engineer
Rupesh
```

Issue with multiple inheritance:

If we call work method on r1 then we cant figure out which work to call,and creates ambiguity will result in compiler error.

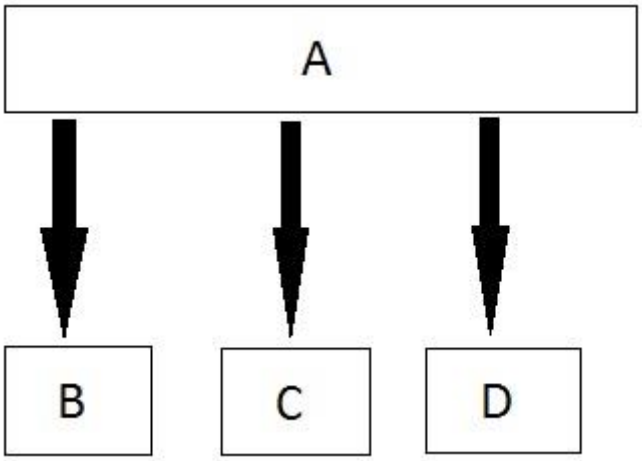
```
int main()
{
    Rupesh r1;
    r1.work(); // Compiler error.
    // Solution to this is
    r1.Engineer::work(); // This will work fine.
    r1.Youtuber::work();// This will work fine.
    static_cast<Engineer>(r1).work();// This also will work fine.
    static_cast<Youtuber>(r1).work();// This also will work fine.

    return 0;
}
```

This issue is not present if we call method which are not common in base classes.

```
int main()
{
    Rupesh r1;
    // Following lines don't have any issue.
    r1.doShooting();
    r1.doProgram();
    return 0;
}
```

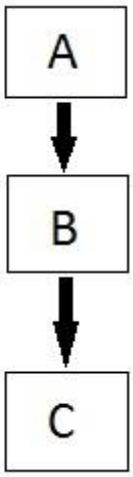
❖ Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



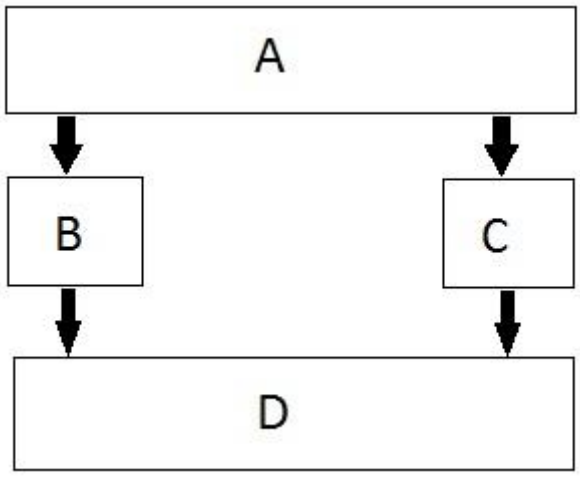
❖ Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



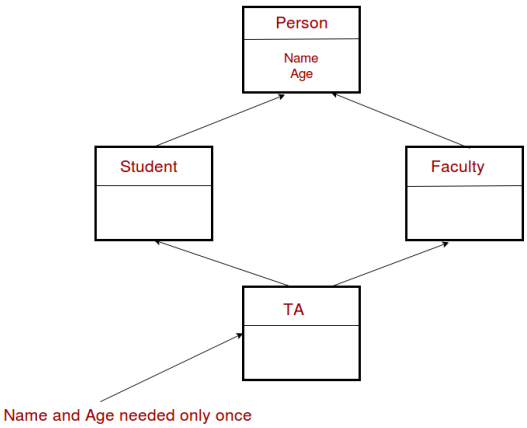
❖ Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities



When we inherit more than one base class in the same derived class and all these base classes also inherit another but same single class (super parent), multiple references of the super parent class become available to the derived class.
So, it becomes unclear to the derived class, which version of the super parent class it should refer to.
e.g:

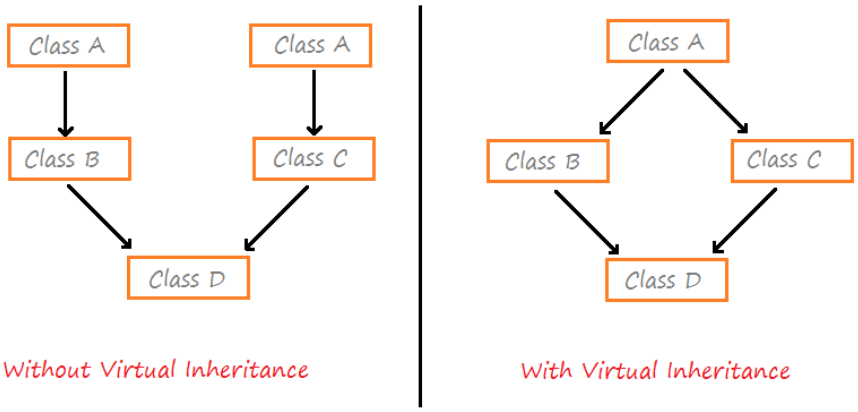
```
//super parent class
class A {
public:
    void name() {
        cout << "This is class A \n";
    }
};

//base class I
class B : public A {};

//base class II
class C : public A {};

//derived class
class D : public B, public C {};

int main()
{
```

```
    D d;
    d.name();

    return 0;
}
```

There is no syntactical error in the above program but still, if we try to compile then it returns the following compilation error:

```
line 24|error: request for member 'name' is ambiguous
```

This is because two instances of class A's name() method is available for class D, one through class B, and the other through class C.



In this case, the compiler gets confused and cannot decide which `name()` method should be refered.
This ambiguity often occurs in the case of multiple inheritances and is popularly known as the **diamond problem** in C++.
To remove this ambiguity, we use virtual inheritance to inherit the super parent.

What is Virtual Inheritance?

Virtual inheritance in C++ is a type of inheritance that ensures that only one copy or instance of the base class's members is inherited by the grandchild derived class.

It is implemented by prefixing the `virtual` keyword in the inheritance statement.
e.g:

```
//base class I
class B : virtual public A {};

//base class II
class C : virtual public A {};
```

❖ Can struct inherit from class or vice a versa?

Yes. Except for their default behavior.

```
class Base
{
};

struct D1 : public Base
```

```
{

};
//Or
struct Base
{
};

class D1 : Base
{
};
```

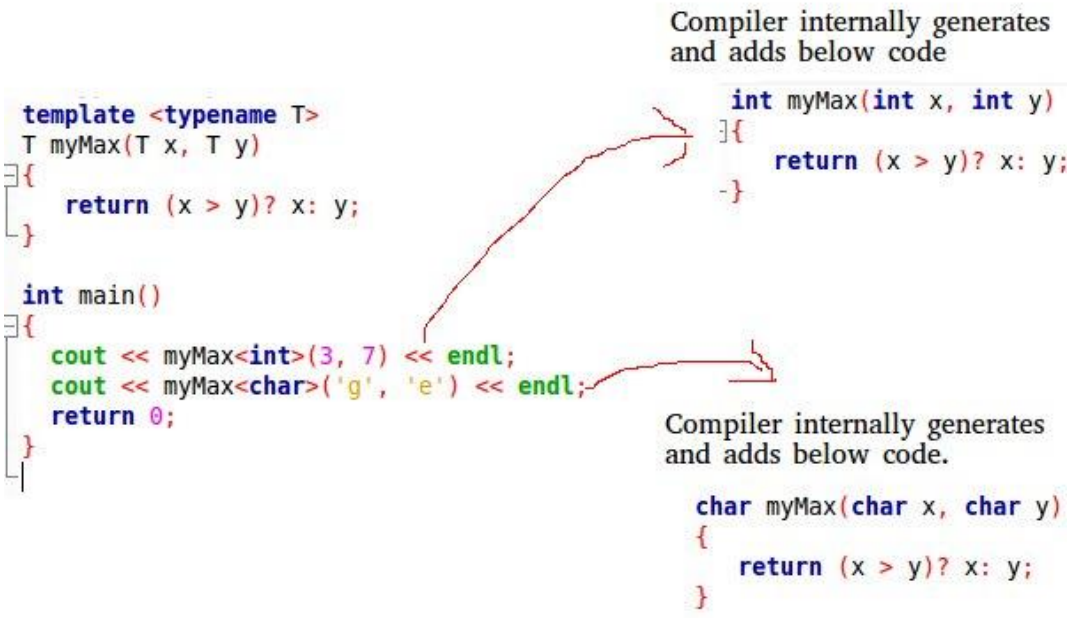
Just default inheritance is private in case of class ad public in case of struct.

• Template

A template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.
C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.

How do templates work?

Templates are expanded at compiler time. This is like macros. The difference is, the compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



Suppose following example,

```
typedef int T;

template <class T>
```

```
class Test
{
	T m_val;
public:
	Test(T val)
	{
		m_val = val;
	}
};

Test < double> t1(5); //Call template with double val = 5.0000

Test <T> t1(5); //Call template with int val = 5
```

❖ Partial template specialization

Partial template specialization is a particular form of class template specialization. Usually used in reference to the C++ programming language, it allows the programmer to specialize only some arguments of a class template, as opposed to explicit specialization, where all the template arguments are provided.

Templates can have more than one parameter type. Some older compilers allow one only to specialize either all or none of the template's parameters. Compilers that support partial specialization allow the programmer to specialize some parameters while leaving the others generic.

Suppose there exists a KeyValuePair class with two template parameters, as follows.

```
template <typename Key, typename Value>
class KeyValuePair {};
```

The following is an example of a class that defines an explicit (or full) template specialization of KeyValuePair by pairing integers with strings. The class type retains the same name as the original version.

```
template <>
class KeyValuePair<int, std::string> {};
```

The next is an example of partial specialization of KeyValuePair with the same name as the original version and one specialized template parameter.

```
template <typename Key>
class KeyValuePair<Key, std::string> {};
```

The next example class KeyStringPair is derived from the original KeyValuePair with a new name, and defines a partial template specialization. In contrast to the explicit specialization above, only the Value template parameter of the superclass is specialized, while the Key template parameter remains generic.

```
template <typename Key>
class KeyStringPair : public KeyValuePair<Key, std::string> {};
```

It does not matter which template parameters are specialized and which remain generic. For instance, the following is also a valid example of a partial specialization of the original KeyValuePair class.

```
template <typename Value>
class IntegerValuePair : public KeyValuePair<int, Value> {};
```

e.g:

// legal: base function template

```
template <typename ReturnType, typename ArgumentType>
ReturnType Foo(ArgumentType arg);

// legal: explicit/full function template specialization
template <>
string Foo<string, char>(char arg) { return "Full"; }

// illegal: partial function template specialization of the return type
//                  function template partial specialization is not allowed
// template <typename ArgumentType>
// void Foo<void, ArgumentType>(ArgumentType arg);

// legal: overloads the base template for a pointer argument type
template <typename ReturnType, typename ArgumentType>
ReturnType Foo(ArgumentType *argPtr) { return "PtrOverload"; }

// legal: base function name reused. Not considered an overload.
template <typename ArgumentType>
string Foo(ArgumentType arg) { return "Return1"; }

// legal: base function name reused. Not considered an overload.
template <typename ReturnType>
ReturnType Foo(char arg) { return "Return2"; }
```

// note: to be compiled in conjunction with the definitions of Foo above

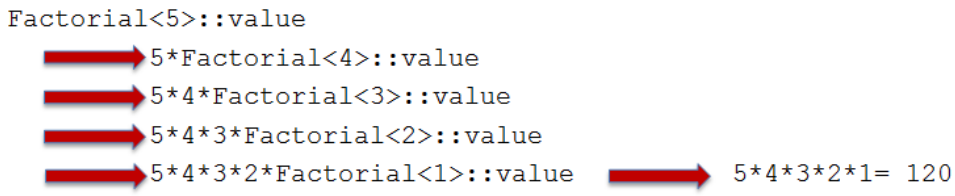
```
int main(int argc, char *argv[])
{
	char c = 'c';
	string r0, r1, r2, r3;
	// let the compiler resolve the call
	r0 = Foo(c);
	// explicitly specify which function to call
	r1 = Foo<string>(c);
	r2 = Foo<string, char>(c);
	r3 = Foo<string, char>(&c);
	// generate output
	std::cout << r0 << " " << r1 << " " << r2 << " " << r3 << std::endl;
	return 0;
}

//expected output:
Return1 Return2 Full PtrOverload
```

❖ Template Metaprogramming in C++

Template metaprogramming (**TMP**) is a metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled. The output of these templates can include compile-time constants, data structures, and complete functions. The use of templates can be thought of as compile-time polymorphism.

```
template <int N>                                                    // (2)
struct Factorial {
    static int const value = N * Factorial<N - 1>::value;
};

template <>                                                         // (3)
struct Factorial<1> {
    static int const value = 1;
};

int main() {

    std::cout << std::endl;

    std::cout << "Factorial<5>::value: " << Factorial<5>::value << std::endl;    // (1)
    std::cout << "Factorial<10>::value: " << Factorial<10>::value << std::endl;

    std::cout << std::endl;

}
```

The call factorial<5>::value in line (1) causes the instantiation of the primary or general template in line (2). During this instantiation, Factorial<4>::value will be instantiated. This recursion will end if the fully specialised class template Factorial<1> kicks in in line (3). Maybe, you like it more pictorial.



Exception Handling in C++

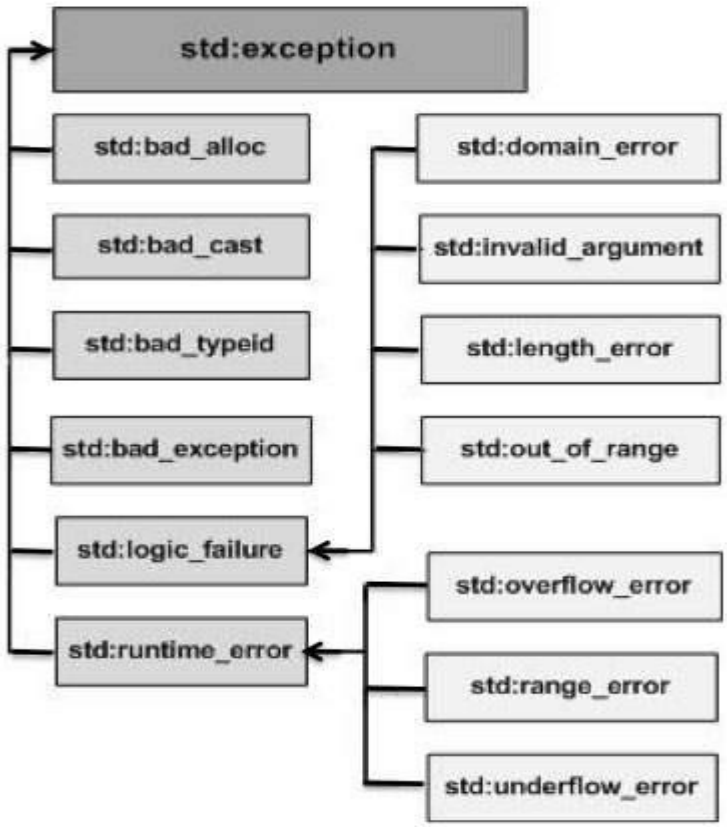
One of the advantages of C++ over C is Exception Handling. C++ provides following specialized keywords for this purpose.
try: represents a block of code that can throw an exception.
catch: represents a block of code that is executed when a particular exception is thrown.
throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

❖ Why Exception Handling?

1) Separation of Error Handling code from Normal Code:
2) Functions/Methods can handle any exceptions they choose:
3) Grouping of Error Types:
4) We perform exception handling so the **normal flow of the application can be maintained even after runtime errors.**

❖ C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs.



❖ Key Notes Exception Handling in C++

1) Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

```
#include <iostream>
```

```
using namespace std;

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}
```

Output:

```
Before try

Inside try

Exception Caught

After catch (Will be executed)
```

2) There is a special catch block called ‘catch all’ catch(…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(…) block will be executed.

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

int main()
{
     try  {
        throw 10;
     }
     catch (char *excp)  {
          cout << "Caught " << excp;
     }
     catch (...)  {
          cout << "Default Exception\n";
     }
     return 0;
}
```

Output:

```
Default Exception
```

3) Implicit type conversion doesn't happen for primitive types. For example, in the following program 'a' is not implicitly converted to int

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

int main()
{
     try  {
        throw 'a';
     }
     catch (int x)  {
          cout << "Caught " << x;
     }
     catch (...)  {
          cout << "Default Exception\n";
     }
     return 0;

}
```

Output:

```
Default Exception
```

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

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

int main()
{
     try  {
        throw 'a';
     }
     catch (int x)  {
          cout << "Caught ";
     }
     return 0;

}
```

Output:

```
terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an

unusual way. Please contact the application's support team for

more information.
```

5)A derived class exception should be caught before a base class exception.

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

class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    // some other stuff
    try {
         // Some monitored code
         throw d;
    }
    catch(Base b) {
          cout<<"Caught Base Exception";
    }
    catch(Derived d) {  //This catch block is NEVER executed
          cout<<"Caught Derived Exception";
    }
    getchar();
    return 0;
}
```

Output:

"Caught Base Exception"

If we change the order of catch statements then both catch statements become reachable. Following is the modified program and it prints "Caught Derived Exception".

6) In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using "throw; "

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

int main()
{
    try {
              try
              {
                     throw 20;
              }
              catch (int n)
              {
                    cout << "Handle Partially ";
                    throw;   //Re-throwing an exception
              }
       }
       catch (int n) {
           cout << "Handle remaining ";
       }
       return 0;
}
```

Output:

```
Handle Partially Handle remaining
```

7) When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to 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;
  }
}
```

Output:

```
Constructor of Test

Destructor of Test

Caught 10
```

❖ Stack Unwinding in C++

The process of removing function entries from function call stack at run time is called Stack Unwinding.

```
#include <iostream>

using namespace std;

// A sample function f1() that throws an int exception
void f1() throw (int) {
  cout<<"\n f1() Start ";
  throw 100;
  cout<<"\n f1() End ";
}

// Another sample function f2() that calls f1()
void f2() throw (int) {
  cout<<"\n f2() Start ";
  f1();
  cout<<"\n f2() End ";
}

// Another sample function f3() that calls f2() and handles exception thrown by f1()
void f3() {
  cout<<"\n f3() Start ";
  try {
    f2();
  }
  catch(int i) {
   cout<<"\n Caught Exception: "<<i;
  }
```

```
  cout<<"\n f3() End";
}

// A driver function to demonstrate Stack Unwinding  process
int main() {
  f3();

  getchar();
  return 0;
}
```

O/p:

```
f3() Start

 f2() Start

 f1() Start

 Caught Exception: 100

 f3() End
```

F1 & F2 function calls removed from stack. Also if there were some local class objects inside f1 & f2 were destroyed in stack unwinding.

❖ Exceptions in destructor

Program will crash if we threw exception from destructor.

Comsider following code,

```
class Excp
{
public:
    ~Excp() { throw 10; }
};
int main()
{
    try {
        Excp e1;
        Excp e2;
    }
    catch (...)
    {
        cout << "Exception handelled..";
    }
}
```

Here when try block over e2 gets destroyed first and it throws ecxeption. Now as a stack unwinding happens e1 also gets destructed before catch block and e1 destructor also throws aother exception. Here program cant handle two exceptions at a time ad calles terminate function.

To avoid this scenario one must handle the in destructor only like,

```
    ~Excp() {
        try {
            throw 10;
        }
        catch (...)
        {
            cout << "Exception handelled..";
        }
```

```
}
```

However this is not adviced.

❖ Exceptions in threads

Will a program crash if one of the threads run into exceptions?

- According to MSDN (http://msdn.microsoft.com/en-us/library/ac9f67ah.aspx), if no appropriate catch handler is found, then the terminate function will be called. terminate by default calls the C runtime function abort though you can override this if necessary by calling set_terminate.
- In a multi-threaded scenario terminate would be called in this secondary thread, which would cause your program to quit - i.e. your main/parent/entry thread would stop immediately and your process killed by the OS.
- An exception can be thrown in one part of a program and caught in a different part. An exception that is not caught will generally cause the program to crash. (More exactly, the thread that throws the exception will crash.

❖ C++ catch blocks - catch exception by value or reference?

Throw by value, catch by reference

Suppose for your example that there is another type MyException which inherits from CustomException and overrides items like an error code.

1. No need to create new copy of Exception again.

In (CustomException e) new object of CustomException is created... so it's constructor will be called while in (CustomException &e) it will just the reference... not new object is created and no constructor will be called

2. Will change error codes
   If a MyException type was thrown your catch block would cause it to be converted to a CustomException instance which would cause the error code to change.

3. Catching by value will *slice* the exception object if the exception is of a derived type to the type which you catch.

4. If you want to modify the object, yet continue exception processing with throw; on the *non-modified* object, then you must make a copy, and one way of doing so is catch by value.

Static and Dynamic Libraries

❖ Key Notes

When a C++ program is compiled, the compiler generates object code. After generating the object code, the compiler also invokes linker. One of the main tasks for linker is to make code of library functions (eg printf(), scanf(), sqrt(), ..etc) available to your program. A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time

Static Linking and Static Libraries is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, .a files in Linux and .lib files in Windows. A lib is a unit of code that is bundled within your application executable.

Following are some important points about static libraries.

1. For a static library, the actual code is **extracted from the library** by the linker and **used to build the final executable** at the point you **compile/build** your application.

2. Each process gets its own copy of the code and data. Where as in case of dynamic libraries it is only code shared, data is specific to each process. For static libraries memory footprints are larger. For example, if all the window system tools were statically linked, several tens of megabytes of RAM would be wasted for a typical user, and the user would be slowed down by a lot of paging.

3. Since library code is connected at compile time, the **final executable has no dependencies on the the library at run time** i.e. no additional run-time loading costs, it means that you don't need to carry along a copy of the library that is being used and you have everything under your control and there is no dependency.

4. In static libraries, once everything is bundled into your application, you don't have to worry that the client will have the right library (and version) available on their system.

5. One drawback of static libraries is, for any **change**(up-gradation) in the static libraries, **you have to recompile the main program** every time.

6. One major advantage of static libraries being preferred even now **"is speed"**. There will be no dynamic querying of symbols in static libraries. Many production line software use static libraries even today.

Dynamic linking and Dynamic Libraries Dynamic Linking doesn't require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run, when both the binary file and the library are in memory. Examples of Dynamic libraries (libraries which are linked at run-time) are, **.so** in Linux and **.dll** in Windows.

A **dll** is a standalone unit of executable code. It is l**oaded in the process only when a call is made into that code**. A dll can be used by multiple applications and loaded in multiple processes, while still having only one copy of the code on the hard drive.

Static libraries increase the size of the code in your binary. They're always loaded and whatever version of the code you compiled with is the version of the code that will run.

Dynamic libraries are stored and versioned separately. It's possible for a version of the dynamic library to be loaded that wasn't the original one that shipped with your code if the update is considered binary compatible with the original version.

Dll pros: can be used to reuse/share code between several products; load in the process memory on demand and can be unloaded when not needed; can be upgraded independently of the rest of the program.

Dll cons: performance impact of the dll loading and code rebasing; versioning problems ("dll hell")

Lib pros: no performance impact as code is always loaded in the process and is not rebased; no versioning problems.

Lib cons: executable/process "bloat" - all the code is in your executable and is loaded upon process start; no reuse/sharing - each product has its own copy of the code.

❖ Creating and Using a Dynamic Link Library (C++)

- ## To create a dynamic link library (DLL) project

1. On the menu bar, choose File, New, Project.
2. In the left pane of the New Project dialog box, expand Installed, Templates, Visual C++, and then select Win32.
3. In the center pane, select Win32 Console Application.
4. Specify a name for the project—for example, MathLibrary—in the Name box. Specify a name for the solution—for example,MathLibraryAndClient—in the Solution name box. Choose the OK button.
5. On the Overview page of the Win32 Application Wizard dialog box, choose the Next button.
6. On the Application Settings page, under Application type, select DLL.
7. Choose the Finish button to create the project.

- ## To add a class to the dynamic link library

1.To create a header file for a new class, on the menu bar, choose Project, Add New Item. In the Add New Item dialog box, in the left pane, select Visual C++. In the center pane, select Header File (.h). Specify a name for the header file—for example, MathLibrary.h—and then choose the Add button. A blank header file is displayed.
2.Replace the contents of the header file with this code:

```
// MathLibrary.h - Contains declaration of Function class
#pragma once

#ifdef MATHLIBRARY_EXPORTS
#define MATHLIBRARY_API __declspec(dllexport)
#else
#define MATHLIBRARY_API __declspec(dllimport)
#endif

namespace MathLibrary
{
    // This class is exported from the MathLibrary.dll
    class Functions
    {
    public:
        // Returns a + b
        static MATHLIBRARY_API double Add(double a, double b);

        // Returns a * b
        static MATHLIBRARY_API double Multiply(double a, double b);
```

```
        // Returns a + (a * b)
        static MATHLIBRARY_API double AddMultiply(double a, double b);
    };
}


// MathClient.cpp : Defines the entry point for the console application.
// Compile by using: cl /EHsc /link MathLibrary.lib MathClient.cpp
```

❖ Exporting from a DLL Using __declspec(dllexport)

You can export data, functions, classes, or class member functions from a DLL using the **__declspec(dllexport)** keyword. **__declspec(dllexport)** adds the export directive to the object file so you do not need to use a .def file.

❖ Import a function from a DLL __declspec(dllimport)

The following code example shows how to use **__declspec(dllimport)** to import function calls from a DLL into an application. Assume that func1 is a function that's in a DLL separate from the executable file that contains the **main** function.

Without **`__declspec(dllimport)`**, given this code:

CCopy

```
int main(void)
{
   func1();
}
```

the compiler generates code that looks like this:

asmCopy

```
call func1
```

and the linker translates the call into something like this:

asmCopy

```
call 0x4000000        ; The address of 'func1'.
```

If func1 exists in another DLL, the linker can't resolve this address directly because it has no way of knowing what the address of func1 is. In 32-bit and 64-bit environments, the linker generates a thunk at a known address. In a 32-bit environment the thunk looks like:

asmCopy

```
0x40000000:    jmp DWORD PTR __imp_func1
```

Here __imp_func1 is the address for the func1 slot in the import address table of the executable file. All these addresses are known to the linker. The loader only has to update the executable file's import address table at load time for everything to work correctly.

That's why using __declspec(dllimport) is better: because the linker doesn't generate a thunk if it's not required. Thunks make the code larger (on RISC systems, it can be several instructions) and can degrade your cache performance. If you tell the compiler the function is in a DLL, it can generate an indirect call for you.

So now this code:

CCopy
```c
__declspec(dllimport) void func1(void);
int main(void)
{
   func1();
}
```

generates this instruction:

asmCopy
```
call DWORD PTR __imp_func1
```

There's no thunk and no jmp instruction, so the code is smaller and faster. You can also get the same effect without __declspec(dllimport) by using whole program optimization.

```c
#include "stdafx.h"
#include <iostream>
#include "MathLibrary.h"

using namespace std;

int main()
{
    double a = 7.4;
    int b = 99;

    cout << "a + b = " <<
        MathLibrary::Functions::Add(a, b) << endl;
    cout << "a * b = " <<
        MathLibrary::Functions::Multiply(a, b) << endl;
    cout << "a + (a * b) = " <<
        MathLibrary::Functions::AddMultiply(a, b) << endl;

    return 0;
}
```

- ## To create an app that references the DLL

1.To create a C++ app that uses the DLL that you just created, on the menu bar, choose File, New, Project.

2.In the left pane of the New Project dialog, expand Installed, Templates, Visual C++, and then select Win32.
3.In the center pane, select Win32 Console Application.
4.Specify a name for the project—for example, MathClient—in the Name box.
5.Choose the drop-down button at the end of the Solution control, and then select Add to Solution from the drop-down list. This adds the new project to the same solution that contains the DLL. Choose the OK button.
6.On the Overview page of the Win32 Application Wizard dialog box, choose the Next button.
7.On the Application Settings page, under Application type, select Console application.
8.Choose the Finish button to create the project.

```
1. include header file.
2. Add lib in dependancy list & give address of the lib file as additional lib
directories.
```

Lambda expressions in C++

- **Basics of lambda**
  - In C++11 and later, a lambda expression—often called a *lambda*—is a convenient way of defining an **anonymous function object**.
  - C++ 11 introduced lambda expression to allow us write an inline function which can be used for short snippets of code that are not going to be reuse and not worth naming. In its simplest form lambda expression can be defined as follows:

```
[ capture clause ] (parameters) -> return-type
{
   definition of method
}
```

  - Generally return-type in lambda expression are evaluated by compiler itself and we don't need to specify that explicitly and -> return-type part can be ignored but in some complex case as in conditional statement, compiler can't make out the return type and we need to specify that.
  - A lambda expression can have more power than an ordinary function by having access to variables from the enclosing scope.
  - We can capture external variables from enclosing scope by three ways :
    Capture by reference
    Capture by value
    Capture by both (mixed capture)

  - Syntax used for capturing variables :
    [&] : capture all external variable by reference
    [=] : capture all external variable by value
    [a, &b] : capture a by value and b by reference

  - when you start to write more complex lambdas you will quickly encounter cases where the return type cannot be deduced by the compiler.
    e.g.

```
void func4(std::vector<double>& v) {
    std::transform(v.begin(), v.end(), v.begin(),
        [](double d) -> double {
            if (d < 0.0001) {
                return 0;
            }
            else {
                return d;
            }
        });
}
```

    Here we **need to specify return type explicitly.**

- **How to capture local variables inside Lambda ?**

❖ Capturing Local Variables by value inside Lambda Function

- To capture the local variables by value, specify their name in capture list i.e.

```
// Local Variables
std::string msg = "Hello";
int counter = 10;
// Defining Lambda function and
// Capturing Local variables by Value
auto func = [msg, counter]() {
    //...
};
```

- Now, the variables specified in capture list will be copied inside lambda by value. Inside lambda they can be accessed but can not be changed, because they are const. To modify the we need to add mutable keyword i.e.

  auto func = [msg, counter] () **mutable** { };

Now the captured variables **can be modified**. But their modification will **not affect value of outer scope variables**, because they are **captured by value**.

❖ Capturing Local Variables by Reference inside Lambda

- To capture the local variables by reference, specify their name in capture list with prefix & i.e.

```
// Local Variables
std::string msg = "Hello";
int counter = 10;
// Defining Lambda function and
// Capturing Local variables by Reference
auto func = [&msg, &counter]() {
    //...
};
```

- Now, the variables specified in capture list will be captured inside lambda by **Reference**. Inside lambda they can be **accessed and their value can also be changed**. Also, their modification will **affect value of outer scope variables**, because they are captured by **Reference**.

❖ Capture All Local Variables from outer scope by Value

- To capture all local variables from outer scope by value, pass "=" in the capture list i.e.

```
// Capturing all Local variables by Value
auto func = [=]() {
    //...
};
```

❖ Capture all local variables from outer scope by Reference

- To capture all local variables from outer scope by Reference, pass "&" in the capture list i.e.
- `// Capturing all Local variables by Reference`

```
auto func = [&]() {
```

```
    //...
};
```

❖ Mixing capturing by value and Reference

- We can also mix the capturing mode of Lambda by passing some local variables by value and some by reference i.e.

```
// Capturing All Local variables by value except a & b, which is
// captured by reference here.
auto func = [=, &a, &b]() mutable {};


struct S { void f(int i); };

void S::f(int i) {
    [&, i] {};      // OK
    [&, &i] {};     // ERROR: i preceded by & when & is the default
    [=, this] {};   // ERROR: this when = is the default
    [=, *this] { }; // OK: captures this by value. See below.
    [i, i] {};      // ERROR: i repeated
}
```

❖ Be-aware of capturing local variables by Reference in Lambda

- If in lambda we are capturing local variables by reference, then we need to make sure that when lambda function is accessed or called, then all the by reference captured local variables are still in scope.
- If lambda will try to access or modify a by reference captured local variable, which is not in scope anymore i.e. which has been destroyed due to stack unwinding, then crash can happen.

• Generalized capture (C++ 14)

- In C++14, you can introduce and initialize new variables in the capture clause, without the need to have those variables exist in the lambda function's enclosing scope.
- The initialization can be expressed as any arbitrary expression; the type of the new variable is deduced from the type produced by the expression.
- This feature lets you capture **move-only variables** (such as **std::unique_ptr**) from the surrounding scope and use them in a lambda.

```
pNums = make_unique<vector<int>>(nums);
//...
auto a = [ptr = move(pNums)]()
  {
     // use ptr
  };
```

• Mutable specification

- Typically, a lambda's function call operator is *const-by-value*, but use of the *mutable keyword cancels this out.*
- It *doesn't produce* mutable data members. The mutable specification *enables the body of a lambda expression* to modify variables that *are captured by value.*
- It means if you want to change passed variables you need to specify it with mutable keyword.

Smart Pointer

Container for raw pointer. Holds raw pointer & release it when got out of scope.

= operator & copy constructor were made private to disable assignment & copy of smart pointer.

*operator were overloaded to get value pointed by smart pointer.

```
template <class T>
class SmPtr
{
	T* data_ptr;
	SmPtr& operator=(SmPtr<T> obj);
	explicit SmPtr(const SmPtr<T>& obj);
public:
	explicit SmPtr(T* ptr)
	{
		data_ptr = ptr;
	}

	~SmPtr()
	{
		delete data_ptr;
	}
	T& operator*()
	{
		return *data_ptr;
	}

	T* operator->()
	{
		return data_ptr;
	}
	void func()
	{
		cout<<"Inside SmPtr";
	}
};
```

- **auto_ptr**

auto_ptr is a smart pointer that manages an object obtained via new expression and deletes that object when auto_ptr itself is destroyed.

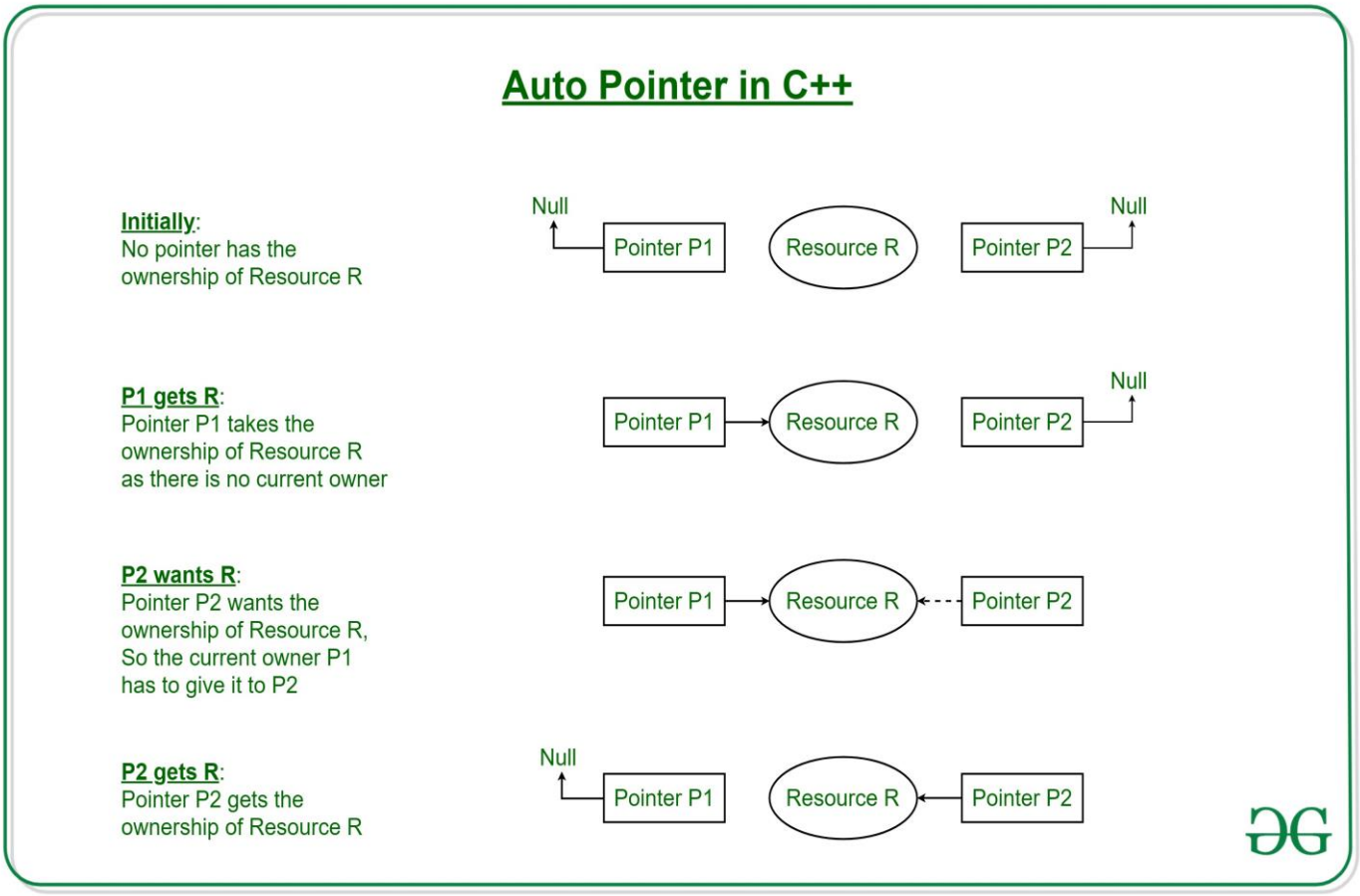
```
int* i = new int;
auto_ptr<int> x(i);
auto_ptr<int> y;

y = x;

cout << x.get() << endl; // Print NULL
cout << y.get() << endl; // Print non-NULL address i
```

❖ Problem with auto_ptr

STL element must be "copy-constructible" and "assignable." In other words, an element must be able to be assigned or copied and the two elements are logically independent. `std::auto_ptr` does not fulfill this requirement.

copying or assigning one auto_ptr to another makes changes to the original, in addition to the obvious changes to the copy. To be more specific, the original object transfers ownership of the pointer to the target, and the pointer in the original object becomes null.

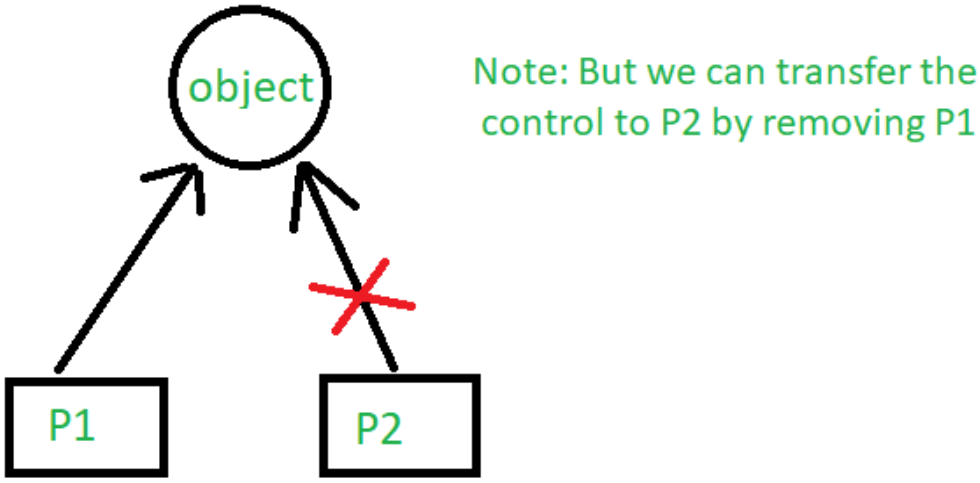
```
class X{};

std::vector<std::auto_ptr<X>> vecX;

vecX.push_back(new X);

std::auto_ptr<X> pX = vecX[0];// vecX[0] is assigned NULL.
```



- ## unique_ptr

A *unique_ptr* is a container for a raw pointer, which the unique_ptr is said to own. A unique_ptr explicitly prevents copying of its contained pointer (as would happen with normal assignment), but the *std::move function can be used to transfer ownership of the contained pointer to another unique_ptr.* A unique_ptr cannot be copied because its copy constructor and assignment operators are explicitly deleted.

unique_ptr stores one pointer only. We can assign a different object by removing the current object from the pointer. First, the unique_pointer is pointing to P1. But, then we remove P1 and assign P2 so the pointer now points to P2.




```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; //Compile error.
std::unique_ptr<int> p3 = std::move(p1); //Transfers ownership. p3 now owns the memory and
p1 is rendered invalid.

p3.reset(); //Deletes the memory.
p1.reset(); //Does nothing.
```

std::unique_ptr::release
Releases ownership of its stored pointer, by returning its value and replacing it with a null pointer. This call does not destroy the managed object, but the unique_ptr object is released from the responsibility of deleting the object.

std::unique_ptr::reset
Destroys the object currently managed by the unique_ptr (if any) and takes ownership of p. If p is a null pointer (such as a default-initialized pointer), the unique_ptr becomes empty, managing no object after the call.

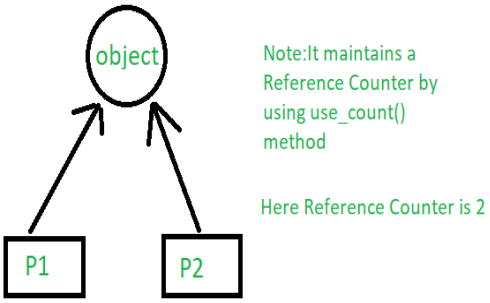
- ## shared_ptr

allows for multiple pointers to point at a given resource. When the very last `shared_ptr` to a resource is destroyed, the resource will be deallocated. For example, this code is perfectly legal:

```
shared_ptr<T> myPtr(new T);// Okay

shared_ptr<T> myOtherPtr = myPtr;// Sure!  Now have two pointers to the resource.
```

Internally, `shared_ptr` uses reference counting to track how many pointers refer to a resource, so you need to be careful not to introduce any reference cycles.

By using shared_ptr more than one pointer can point to this one object at a time and it'll maintain a **Reference Counter** using **use_count() method.**



In short:

- Use *unique_ptr* when you want a single pointer to an object that will be reclaimed when that single pointer is destroyed.
- Use *shared_ptr* when you want multiple pointers to the same resource.

❖ Difference between std::auto_ptr &std::unique_ptr

```
1.

std::auto_ptr<int> a(newint(10)), b;

b = a;//implicitly transfers ownership


std::unique_ptr<int> a(newint(10)), b;

b = std::move(a);//ownership must be transferred explicitly
```

2. As for other differences, unique_ptr can handle arrays correctly (it will call delete[], while auto_ptr will attempt to call delete.

• Make_shared in shared_ptr

Prerequisites:

Basics of Smart Pointer

Overview:

Let's refer below sample code:

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

int main()
{
    int* p = new int;
    /*code implementation
    .....
    .....
    .....
    */
    delete p;
    p = nullptr;

    return 0;
}
```

We must release the heap allocated memory by our own once use is over. There is fair chance of forgetting to do so and hence smart pointer has been introduced. Updated code will be as below:

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

int main()
{
    int* p = new int;

    shared_ptr<int> ptr(p);

    return 0;
}
```

We just included memory header file for usage of shared_ptr and created object of template class shared_ptr. Now, we don't have to worry about deallocation of heap memory, and it will be taken care automatically.

But can you predict on issue in above example? Let's take another example as below:

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

//Defined sample class with class member functions as inline just for demo purpose.
class clsTest
{
    public:
        clsTest() {cout << "Object Created\n";}
        clsTest(const clsTest& obj) {cout << "Object Copied\n";}
        ~clsTest() {cout << "Object Destroyed\n";}
        void display() {cout << "Hi from object\n";}
};

int main()
{
    clsTest* pTest = new clsTest;
    shared_ptr<clsTest> ptr(pTest);
    ptr->display();

    return 0;
}
```

It is perfect and would generate output as below:

```
Object Created
Hi from object
Object Destroyed
```

We can observe that object is destroyed when scope is over. Now let's discuss about tricky part. We decided to get rid of raw pointer (allocation on heap using new operator) so that we don't have to worry about delete, but it is still there. One can play with raw pointer and it could be dangerous. Refer below code:

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

//Defined sample class with class member functions as inline just for demo purpose.
class clsTest
{
    public:
        clsTest() {cout << "Object Created\n";}
        clsTest(const clsTest& obj) {cout << "Object Copied\n";}
        ~clsTest() {cout << "Object Destroyed\n";}
        void display() {cout << "Hi from object\n";}
};

int main()
{
    clsTest* pTest = new clsTest;
    shared_ptr<clsTest> ptr(pTest);

    /*code implementation
    .....
    .....
    .....
    */

    delete pTest;
    pTest = nullptr;

    /*code implementation
    .....
    .....
    .....
    */

    ptr->display();

    return 0;
}
```

As we can see above, once object of shared pointer is created, there can be N number of code lines and by mistake some one deallocates memory of raw pointer using delete operator. While calling display method, ptr is no longer

valid as memory it was pointing to has already been deallocated. Also deleted memory location pTest is again deleted once scope of shared pointer ptr is over. This is undefined behavior and output is dependent on compiler implementation. But obviously we are not happy with the scenario we have. Can we have any better choice?

Solution:

Yes, you guessed it correctly. make shared comes to our rescue. Refer below code:

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

//Defined sample class with class member functions as inline just for demo purpose.
class clsTest
{
    public:
        clsTest() {cout << "Object Created\n";}
        clsTest(const clsTest& obj) {cout << "Object Copied\n";}
        ~clsTest() {cout << "Object Destroyed\n";}
        void display() {cout << "Hi from object\n";}
};

int main()
{
    shared_ptr<clsTest> ptr = make_shared<clsTest>();
    ptr->display();

    return 0;
}
```

There is no involvement of new operator as seen above and its output is as below:

```
Object Created
Hi from object
Object Destroyed
```

Apart from this, there is one more advantage of using make shared. Let's discuss that as well. For this, refer updated code below:

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

//operator overloaded function new to verify the dynamic allocation
void* operator new(size_t size)
{

    cout << "Memory allocation on heap for " << size << " bytes" << endl;
    void* p = malloc(size);//this is memory allocation from heap
    return p;
}
//operator overloaded function delete to verify the dynamic deallocation
void operator delete(void* p)
{

    cout << "Memory deallocation " << endl;
    free(p);//this is memory deallocation from heap
}
//Defined sample class with class member functions as inline just for demo purpose.
class clsTest
{
    public:
        clsTest() {/*cout << "Object Created\n";*/}
        clsTest(const clsTest& obj) {/*cout << "Object Copied\n";*/}
        ~clsTest() {/*cout << "Object Destroyed\n";*/}
        void display() {/*cout << "Hi from object\n";*/}
};
int main()
{
    {
        shared_ptr<clsTest> ptr = make_shared<clsTest>();
        ptr->display();
    }
    cout << "----------------------------------------------------------------\n";
    {
        clsTest* p = new clsTest;
        shared_ptr<clsTest> ptr(p);
        ptr->display();
    }

    return 0;
}
```

We just commented cout statement from class member function (to avoid messy output by so many cout statements) and added global overloaded new and delete operators.

This code has below output:

```
Memory allocation on heap for 24 bytes
Memory deallocation
------------------------------------------------------------------
Memory allocation on heap for 1 bytes
Memory allocation on heap for 24 bytes
Memory deallocation
Memory deallocation
```

Output clearly suggests that in case of make shared, there will be only one allocation on heap whereas raw pointer method of smart pointer initiation requires two memory allocation on heap. As heap allocation is costlier operation, we can conclude that shared pointer using make shared improves performance compared to smart pointer utilizing raw pointer.

Please note that make shared in above example took only 24 bytes and not 24+1 as raw pointer memory (empty class) is adjusted within the make shared memory. Also, these 24 bytes for smart pointer (shared pointer) is also compiler implementation dependent.

Let's change class which takes almost 40 bytes due to 10 sized array of integer type and observe the output:

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

//operator overloaded function new to verify the dynamic allocation
void* operator new(size_t size)
{

    cout << "Memory allocation on heap for " << size << " bytes" << endl;
    void* p = malloc(size);//this is memory allocation from heap
    return p;
}
//operator overloaded function delete to verify the dynamic deallocation
void operator delete(void* p)
{

    cout << "Memory deallocation " << endl;
    free(p);//this is memory deallocation from heap
}
//Defined sample class with class member functions as inline just for demo purpose.
class clsTest
{
    int intArr[10];
};
int main()
{
    {
        shared_ptr<clsTest> ptr = make_shared<clsTest>();
    }
    cout << "-----------------------------------------------------------------\n";
    {
        clsTest* p = new clsTest;
        shared_ptr<clsTest> ptr(p);
    }

    return 0;
}
```

Output:

```
Memory allocation on heap for 56 bytes
Memory deallocation
------------------------------------------------------------------
Memory allocation on heap for 40 bytes
Memory allocation on heap for 24 bytes
Memory deallocation
Memory deallocation
```

Here, make shared took more memory (56 > 40 + 24) but fact is that all memory bytes in case of make shared happens in one go i.e., only one heap allocation. This concludes that makes make shared is always faster compared to raw pointer used in shared pointer approach due to only one heap allocation requirement.

Having said that make shared is better choice in terms of performance, one cannot use the same in following scenarios.

1. Custom deleter can not be provided to make shared and one must opt for raw pointer method in this case.
2. If constructor is private like singleton design pattern.

Above two scenarios do not work with make shared.

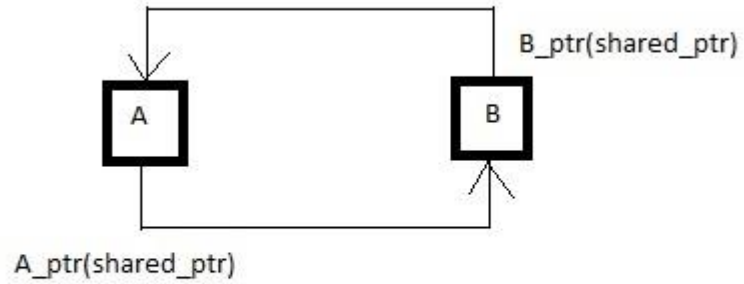
Feel free to share your valuable feedback for this article.

- **weak_ptr**

It's much more ***similar to shared_ptr except it'll not maintain a Reference Counter***. In this case, a pointer will not have a stronghold on the object. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a **Deadlock.**

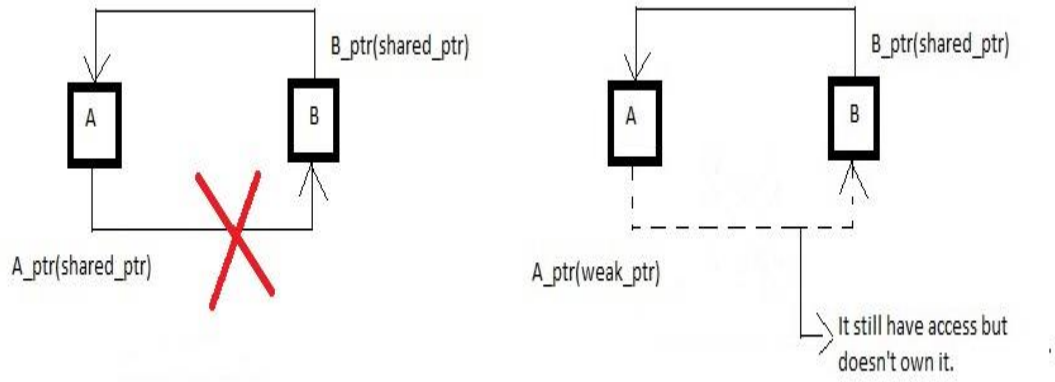
A weak_ptr is created as a copy of shared_ptr. It provides access to an object that is owned by one or more shared_ptr instances but does not participate in reference counting. The existence or destruction of weak_ptr has no effect on the shared_ptr or its other copies. It is required in some cases to break circular references between shared_ptr instances.

Cyclic Dependency (Problems with shared_ptr): Let's consider a scenario where we have two classes A and B, both have pointers to other classes. So, it's always like A is pointing to B and B is pointing to A. Hence, use_count will never reach zero and they never get deleted.



Circular Reference

This is the reason we use **weak pointers**(weak_ptr) as they are not reference counted. So, the class in which weak_ptr is declared doesn't have a stronghold of it i.e. the ownership isn't shared, but they can have access to these objects.



```
std::weak_ptr<int> gw;

void observe()
{
    std::cout << "gw.use_count() == " << gw.use_count() << "; ";
    // we have to make a copy of shared pointer before usage:
    if (std::shared_ptr<int> spt = gw.lock()) {
        std::cout << "*spt == " << *spt << '\n';
    }
    else {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;
        observe();
    }
    observe();
}
```

```
gw.use_count() == 1; *spt == 42
gw.use_count() == 0; gw is expired
```

NOTES:

1. If we say ***unique_ptr*** is for unique ownership and ***shared_ptr*** is for shared ownership then ***weak_ptr*** is for non-ownership smart pointer.
2. It actually reference to an object which is managed by ***shared_ptr.***
3. A ***weak_ptr*** is created as a ***copy of shared_ptr.***
4. ***We have to convert weak_ptr to shared_ptr*** in order to use the managed object.
5. It is used to ***remove cyclic dependency*** between shared_ptr.

e.g:

```
int main()
{

    std::shared_ptr<int> shared = std::make_shared<int>(100);
    cout << "Ref Count of Shared : " << shared.use_count() << endl;
    std::weak_ptr<int> weak = shared;
    cout << "Ref Count of Shared : " << shared.use_count() << endl;
    cout << "Ref Count of weak : " << weak.use_count() << endl;
    if (std::shared_ptr<int> shared1 = weak.lock())
    {
        *shared = 200;
        cout << "Ref Count of Shared : " << shared.use_count() << endl;
        cout << "Ref Count of weak : " << weak.use_count() << endl;
       // Note here ref count of shared and weak ptr is 2.At this location weak pointer is
       equivalent to normal shared pointer.
    }
    else
        cout << "Shared ptr is not valid now. " << endl;
    weak.reset();
    cout << "Ref Count of Shared : " << shared.use_count() << endl;
    cout << "Ref Count of weak : " << weak.use_count() << endl;
```

```
    // Now again we cant get lock as we reset the weak ptr unless we again assign it with
any shared ptr
    weak = shared;
    if (std::shared_ptr<int> shared1 = weak.lock())
    {
        *shared = 300;
        cout << "Ref Count of Shared : " << shared.use_count() << endl;
        cout << "Ref Count of weak : " << weak.use_count() << endl;
    }
    return 0;
}
```

O/P:

Ref Count of Shared : 1

Ref Count of Shared : 1

Ref Count of weak : 1

Ref Count of Shared : 2

Ref Count of weak : 2

Ref Count of Shared : 1

Ref Count of weak : 0

Ref Count of Shared : 2

Ref Count of weak : 2

e.g : To understand cyclic dependency.

```
class Daughter;
class Son;

class Mother
{
    shared_ptr<Daughter> myDaughter;
    shared_ptr<Son> mySon;
public:
    Mother() {}
    ~Mother() { cout << "Mother gone."; }
    void setSon(shared_ptr<Son> son) { mySon = son; }
    void setDaughter(shared_ptr<Daughter> daughter) { myDaughter = daughter; }
};
class Daughter
{
    shared_ptr<Mother> myMother;
public:
    Daughter(shared_ptr<Mother> mother) :myMother(mother){}
    ~Daughter() { cout << "Daughter gone."; }
};
class Son
{
    shared_ptr<Mother> myMother;
public:
    Son(shared_ptr<Mother> mother):myMother(mother) {}
    ~Son() { cout << "Son gone."; }
};


int main()
```

```
{
    std::shared_ptr<Mother> mother = std::make_shared<Mother>();
    std::shared_ptr<Daughter> daughter = std::shared_ptr<Daughter>(new Daughter(mother));
    std::shared_ptr<Son> son = std::shared_ptr<Son>(new Son(mother));
    mother->setDaughter(daughter);
    mother->setSon(son);

    return 0;
}
```

O/P:
Nothing gets deleted as mother is dependant on son & daughter as it holds their shared pointer. And daughter & son depends on mother as they both holds mothers shared pointer.
So no one gets destructed.

Here weak pointers come into action,If we make mother class like ,

```
class Mother
{
    weak_ptr<Daughter> myDaughter; // Weak pointer
    weak_ptr<Son> mySon; // Weak pointer
….
}
```

As they are not tightly coupled mother object can get deleted, so the child objects also.

Functors in C++

Please note that the title is **Functors** (Not Functions)!!
Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

Functors are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs in a scenario like following:

```
int increment(int x) { return (x + 1); }

int main()
{
    int arr[] = { 1, 2, 3, 4, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Apply increment to all elements of arr[] and store the modified elements
    // back in arr[]
    transform(arr, arr + n, arr, increment);
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

This code snippet adds only one value to the contents of the arr[]. Now suppose, that we want to add 5 to contents of arr[].

A **functor** (or function object) is a **C++ class that acts like a function**. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the operator().

The line,

```
MyFunctor(10);
```

Is same as

```
MyFunctor.operator()(10);
```

```
// A Functor
class increment
{
private:
    int num;
public:
    increment(int n) : num(n) {  }

    // This operator overloading enables calling operator function () on objects of
increment
    int operator () (int arr_num) const {
        return num + arr_num;
    }
};

// Driver code
int main()
{
    int arr[] = { 1, 2, 3, 4, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int to_add = 5;

    transform(arr, arr + n, arr, increment(to_add));

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

The line,

```
transform(arr, arr + n, arr, increment(to_add));
```

is the same as writing below two lines,

// Creating object of increment

```
increment obj(to_add);
```

// Calling () on object

```
transform(arr, arr + n, arr, obj);
```

```
#include <iostream>
#include <thread>


class Me {
public:
    bool isLearning;
    void operator()(bool startLearning) {
        isLearning = startLearning;
    }
};

int main() {
```

```
    Me m;
    std::thread t1(std::ref(m), true); // Check param passing
    t1.join();
    std::cout << m.isLearning << std::endl;
}
```

Memory leak in C++ and How to avoid it?

Memory leakage occurs in C++ when programmers allocates memory by using new keyword and forgets to deallocate the memory by using delete() function or delete[] operator. One of the most memory leakage occurs in C++ by using wrong delete operator.
The delete operator should be used to free a single allocated memory space, whereas the delete [] operator should be used to free an array of data values.

Disadvantage with memory leakage:
If a program has memory leaks, then its memory usage is satirically increasing since all systems have limited amount of memory and memory is costly. Hence it will create problems.

Example of memory leakage in C++

```
// function with memory leak
void func_to_show_mem_leak()
{
    int* ptr = new int(5);
    // body
    // return without deallocating ptr
    return;
}
```

How to avoid Memory Leak?

- Instead of managing memory manually, try to use smart pointers where applicable.
- use std::string instead of char *. The std::string class handles all memory management internally, and it's fast and well-optimized.
- *Never use a raw pointer* unless it's to interface with an older lib.
- The best way to avoid memory leaks in C++ is to have as few new/delete calls at the program level as possible – ideally NONE. Anything that requires dynamic memory should be buried inside an RAII object that releases the memory when it goes out of scope. RAII allocate memory in constructor and release it in destructor, so that memory is guaranteed to be deallocated when the variable leave the current scope.
- Allocate memory by new keyword and deallocate memory by delete keyword and write all code between them.

Copy-on-write

Intent

Achieve lazy copy optimization. Like lazy initialization, do the work just when you need because of efficiency.

Also Known As

- COW (copy-on-write)
- Lazy copy

Motivation

Copying an object can sometimes cause a performance penalty. If objects are frequently copied but infrequently modified later, **copy-on-write** can provide significant optimization. To implement copy-on-write, a smart pointer to the real content is used to encapsulate the object's value, and on each modification an object reference count is checked; if the object is referenced more than once, a copy of the content is created before modification.

```
template <class T>
class CowPtr
{
public:
    typedef std::shared_ptr<T> RefPtr;

private:
    RefPtr m_sp;

    void detach()
    {
        T* tmp = m_sp.get();
        if (!(tmp == 0 || m_sp.unique())) {
            m_sp = RefPtr(new T(*tmp));
        }
    }

public:
    CowPtr(T* t)
        : m_sp(t)
    {}
    CowPtr(const RefPtr& refptr)
        : m_sp(refptr)
    {}
    const T& operator*() const
    {
        return *m_sp;
    }
    T& operator*()
    {
        detach();
        return *m_sp;
    }
    const T* operator->() const
    {
        return m_sp.operator->();
    }
```

```
    T* operator->()
    {
        detach();
        return m_sp.operator->();
    }
};


int main()
{
    std::shared_ptr<string> sp (new string("Hello"));
    CowPtr<string> s2(sp); // Till this point string address is same.
    cout << *s2; // Here separate copy is made for s2 string
    return 0;
}
```

e.g:

Disadvantages

```
int main()
{
    std::shared_ptr<string> sp (new string("Hello"));
    char& c = sp->operator[](4); // Non-const detachment does nothing here
    CowPtr<string> s2(sp); // Till this point string address is same.
    c = '!'; // Uh-oh We changed char of s1 & S2 both as they share same memory.
    cout << *s2; // This will print "Hell!". But programmer is anot expecting this...
    return 0;
}
```

The intention of the last line is to modify the original string `s1`, not the copy, but as a side effect `s2` is also accidentally modified.

e.g.2:

```
class MyString
{
private:
    char* m_str = nullptr;
public:
    MyString(const char* strIn)
    {
        if (strIn != nullptr)
        {
            int len = strlen(strIn);
            m_str = new char[len + 1] ();
            strcpy_s(m_str,len+1, strIn);

        }
    }

    MyString(const MyString& strIn)
    {
```

```
        if (strIn.m_str != nullptr)
        {
            m_str = strIn.m_str; // Just do shallow copy.
        }
    }
    MyString(MyString&& strIn);

    MyString& operator=(const MyString& strIn)
    {
        if (this != &strIn && m_str != strIn.m_str)
        {
            m_str = strIn.m_str;
        }
        return *this;
    }
    MyString& operator+(const char* strIn)
    {
        if (strIn != nullptr)
        {
        size_t str1size = strlen(m_str);
        size_t str2size = strlen(strIn);
        size_t size = sizeof(char) * (str1size + str2size + 1);

        char* c = (char* )malloc(size);
        memcpy(c, m_str, str1size);
        memcpy(c + str1size, strIn, str2size);
        c[str1size + str2size] = '\0';
        m_str = c;
       }

        return *this;
    }

    ~MyString() { delete[] m_str; }
};

int main()
{
    const char* str = "abc";
    MyString m1(str);
    MyString m2 = m1; // Till this point char* address is same.
    const char* newstr = "xyz";
    m2 = m2 + newstr; // Here we assign new memory for s2.
    if(*m2 != nullptr)
        cout << *m2;

    return 0;
}
```

- ## Custom Deleter

❖ Why custom delete?

If we need to call some more functions before deleting the pointer. There would e a situations where we would like to delete more than just deleting pointer. Suppose we have a class which allocated many resources and it defines one function release which releases those resources in custom way before deleting the pointer. If we forgot to call release we will leak memory here.

```
class A
{
```

```
public:
    ~A() { /*Delete obj.....*/ }
    void release() { /*Delete some extra things...*/ }
};

int main()
{

    A* ptr = new A();
    //......
    ptr->release();
    delete ptr;

    // Or we can can create like this but here also we need to call release function to
release the resources.

   shared_ptr<A> sp = shared_ptr<A>(new A());
   sp->release();

}
```

To avoid this memory leak either we need to call release function before every location where we are deleting the pointer or we can call it from destructor.But sometime we cant call that function from destructor because its legacy code and we don't want to make chage in it or many other cliets are using the same code. So here custom deleter come to rescue,

❖ Custom deleter for shared_ptr:

Specified in constructor of shared pointer.

```
std::shared_ptr<A> spA(new A(), &deleteA);
```

Deleter can be :

Function pointer:

```
void deleteA(A* ptr) { ptr->release(); delete ptr;}

int main()
{
    shared_ptr<A> spA (new A(), &deleteA);

    // Now no need to delete pointer and no need to release resources too.

}
```

Lambda functions:

```
shared_ptr<A> spA(new A(), [](A* ptr) {ptr->release(); delete ptr;});
```

Functor:

```
class Del
{
public:
    void operator()(A* ptr) {
        ptr->release();
        delete ptr;
    }
};

shared_ptr<A> spA(new A(), Del());
```

❖ Custom deleter for unique_ptr:

- Need to specify in constructor.
- Deleter type will be part of unique pointer.

Function Pointer:

```
void deleteA(A* ptr) { ptr->release(); delete ptr; }
unique_ptr<A, decltype(deleteA)> up(new A(), &deleteA);
```

Lambda function:

```
auto deleter = [](MyType*) { ... }
std::unique_ptr<MyType, decltype(deleter) >> u1(new MyType(), deleter);
```

❖ Restrictions that come with custom deleter

- ### Can’t use make_shared with shared_ptr

Unfortunately, you can pass a custom deleter only in the constructor of shared_ptr there is no way to use make_shared. This might be a bit of disadvantage
One can use allocate_shared and custom allocator.

- ### Can’t use make_unique with unique_ptr

Similarly as with shared_ptr you can pass a custom deleter only in the constructor of unique_ptr and thus you cannot use make_unique.

Memory Leak and Corruption

A **memory leak** occurs when memory is allocated but never freed.
This causes wasting memory, and eventually leads to a potentially fatal out-of-memory.

A **memory corruption** occurs when the program writes data to the wrong memory location, overwriting the data that was there, and failing to update the intended location of memory.

problems caused by pointers

- Copying a pointer does not copy the corresponding object, leading to surprises if two pointers inadvertently points to the same object.
- Destroying a pointer does not destroy its object, leading to memory leaks.
- Destroying an object without destroying a pointer to it leads to a **dangling pointer**, which causes undefined behavior if the program uses the pointer.
- Creating a pointer without initializing it leaves the pointer unbound, which also causes undefined behavior if the program uses it. This uninitialized pointer is sometimes called **wild pointer.**

Reasons of Crash

- Un-initialized pointer operation - invalid access resulting with an attempt to read or write using a NULL pointer.

- Invalid array indexing - out of bound array indexing.
- Illegal stack operation - a program passes a pointer of the wrong type to a function.
- Accessing an illegal address.
- Infinite loop - invalid array indexing when the loop index exceeds the array bounds and corrupts memory.
- Invalid object pointer - invoking a method for an illegal object pointer.
- Corruption of the v-table pointer.
- Not checking for memory allocation failure.

How you find memory leaks?

There many ways to find memory leaks, One of the ways is by using MFC class. And another way is using Purify tools...

CMemorState is a MFC class by which we can find the memory leaks. Below is a sample code to find the same.

```
#ifdef _DEBUG
CMemoryState oldState, newState, diffState;
oldState.Checkpoint();
#endif
int* a = new int[10];
#ifdef _DEBUG
newState.Checkpoint();
if(diffState.Difference(oldState, newState))
{
TRACE0("Memory Leaked");
}
#endif
```

What Is Team Foundation Server?

- Team Foundation Server (Microsoft TFS) helps manage teams and their code.

Configuring:

To Start working with TFS, first we will have to connect to TFS server and setup the initials. On visual studio menu, you should see a menu named 'Team', which will include only one option titled 'Connect To Team Foundation Server'.

Browse Server Contents:

Select location where we can save all server files inside your team foundation server project.

Adding/Deleting Files/Directories To Source Control

To add one/more file(s)(or directories), just use the 'Add Items To Folder' option to do so..

Similarly you can delete files/folders from server using the 'delete' option and then finally checking in the pending changes.

Edit File:

To Edit a file, you should first use the option 'Checkout for edit' to acknowledge server that you are going to edit this file.

Merging And Branching:

TFS has merging and branching facility as well. If your local version doesn't fit with server version, in short, if any conflicts occurs, you can either manually merge them in merge tool or you can command tfs to do auto merge.

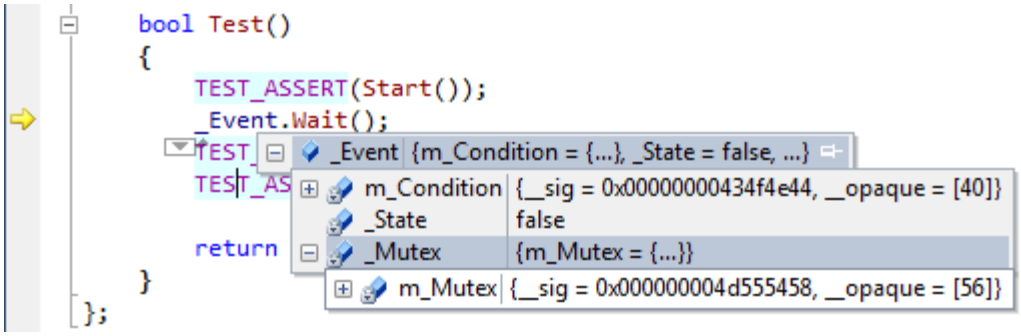
If you have to work on several files and all are related to a single feature, its best to create a branch of the project or code file(s) and work on them. After finishing your task. you can again use the TFS merge command with the main branch and it will automatically incorporate those changes itself.

Shelve Set:

Team foundation server has this feature, which allows you to save your local edit as a separate item in server, which can be retrieved and reviewed by other developers of your team. After approval suggestions, it can be deleted or check in to the main version control stream.

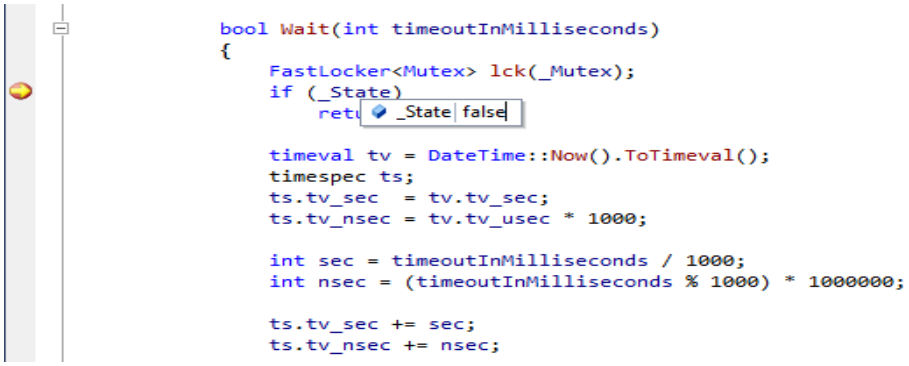
Powerful debugging tricks

1. Hover mouse to evaluate expression



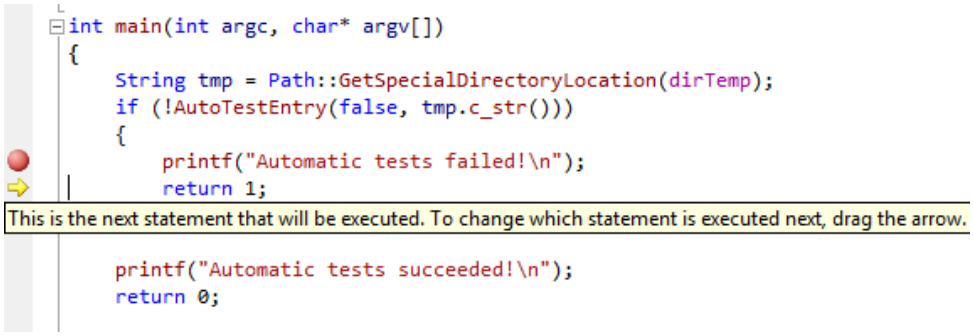
classes and structures will be expanded with one click, allowing to find the field you need quickly and conveniently.

2. Change values on-the-fly



Just hover the mouse over a variable, double-click at the value and type in the new one!

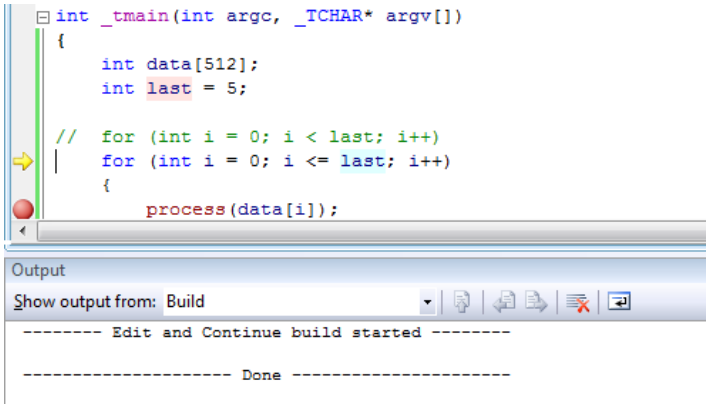
3. Set next statement



just drag the yellow statement marker to the line you want to be executed next, like the function that has

just failed, and then simply step in.

4. Edit and continue



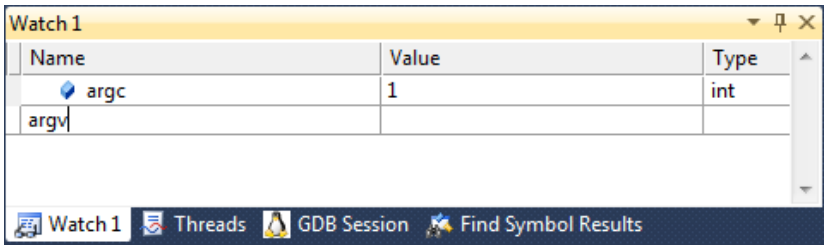
Visual Studio will modify your program and continue debugging with no need to restart.

Limitations:

it won't work for 64-bit code.

edit-and-continue changes should be local, i.e. within one method. If you change the method signature, add new methods or classes, you'll have to restart the app

5. A convenient watch window

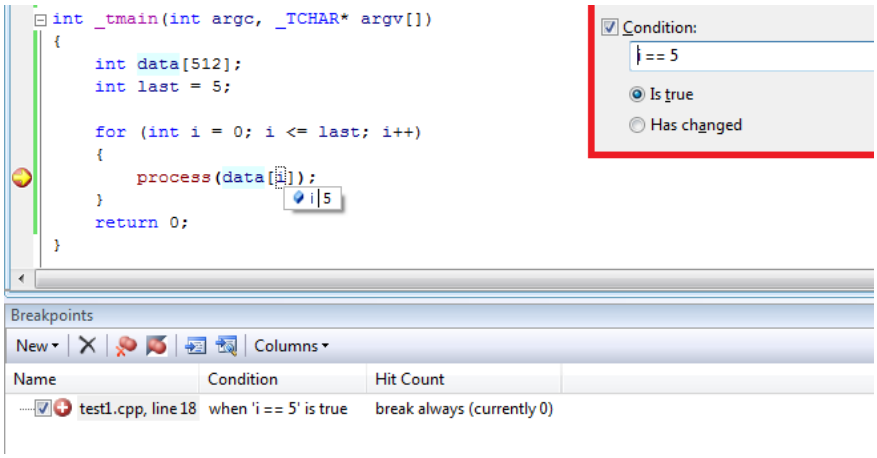


You can add lookup for particular variable in watch windows.

6. Breakpoints

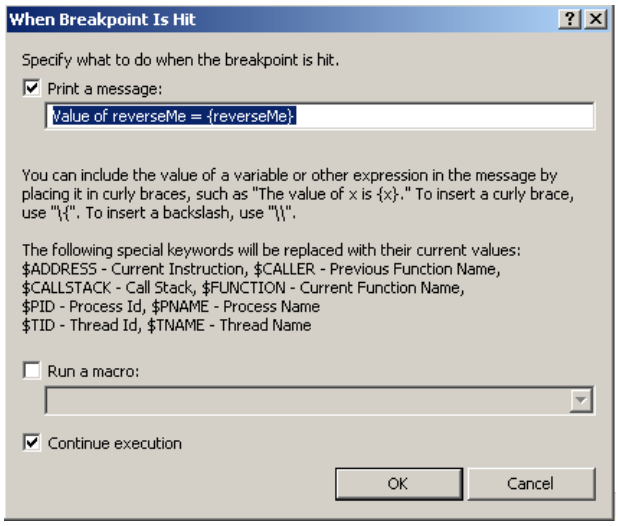
Conditional breakpoints

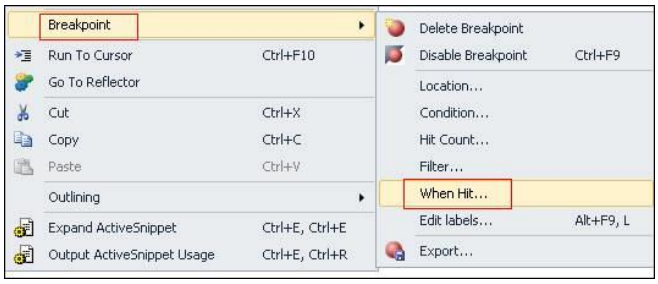
If you're trying to reproduce a rare event and getting too many false positives with your breakpoints, you can easily make them conditional! Simply specify the condition for a breakpoint and Visual Studio will automatically ignore the breakpoint when the condition does not hold.

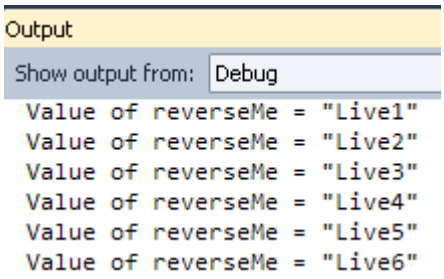


Trace Points

Sometimes we want to observe the value of one or more variables each time a particular line of code is executed. Doing this by setting a normal breakpoint can be very time consuming. So we usually useConsole.WriteLine to print the value. Instead if it's a temporary check using TracePoints is better.



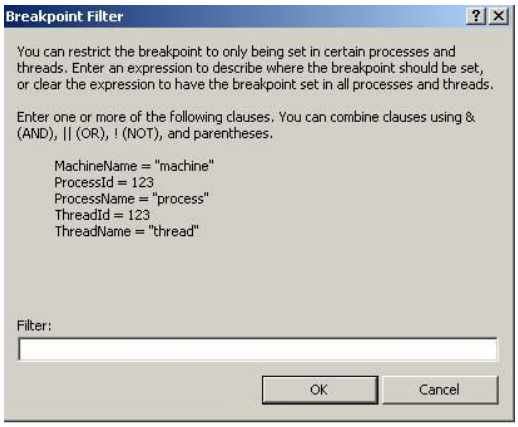




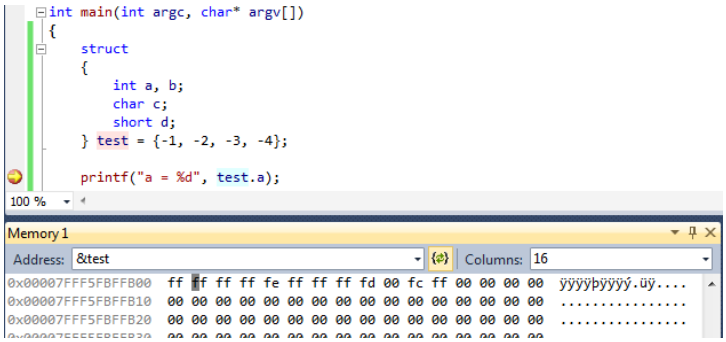
Filter

Useful for multi threaded applications. If multiple threads are calling the same function, you can use filter to specify on which thread should the breakpoint be hit.

Right Click -> Breakpoint -> Filter



7. Memory window



Some bugs are caused by incorrect structure definitions, missing alignment attributes, etc. Seeing the raw memory contents simplifies locating and fixing those bugs.

8. CALL STACK

The Call Stack displays the entire tree of function calls leading to the current function call. Can help you trace back the culprit!

Miscellaneous

- Using null pointer will always result in a crash, right? WRONG!

```cpp
//% It's a fallacy that using nullptr will always result in a crash
//% Here is C++ code which is using nullptr to call a class method
//% Member 'methods' ALWAYS exist, whether the pointer is null or not
//% However, member 'variable' DONT exist unless we create an object
//# [Sanjay Vyas]

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

class Person {
public:
  int id;
  string name;

  Person(int id, string name) : id(id), name(name) {}
  void greet() { cout << "Hello there!" << endl; }
  void print() { cout << id << " " << name << endl; }
};

int main() {
  Person *p = nullptr;
  p->greet();   //> SURPRISE!! Prints → Hello there!
  p->print();   //! 💀 Crash! Boom! Bang!

  return 0;
}
```

C++ 11 Features

- Multithreading
- Smart Pointers
    1. unique_ptr
    2. Shared_ptr
    3. weak_ptr
- Move semantics
- Lambda functions included
- auto and decltype added

C++ 14 features

- Generalized Lambdas

```
// Declare a generalized lambda and store it in sum
auto sum = [](auto a, auto b) {
    return a + b;
};

// Find sum of two integers
cout << sum(1, 6) << endl;

// Find sum of two floating numbers
cout << sum(1.0, 5.6) << endl;
```

In **C++11** this will not compile and give **error** like "**Auto not allowed in lambda parameter**".

- Reader-Writer Locks
- constexpr included
- Return type deductions extended to all functions

C++ 17 features

- The file system library and Network concepts included
- Improved Lambdas
- Fold Expressions included
- Initializers in if and switch statements
- Concurrent and Parallel algorithms in Standard Template Library(STL)
  Concurrency::parallel_for_each
- Nested Namespaces
- Transactional memory
- Inline Variables
- Optional header file
- Class Template argument deduction(CTAD)