Midwestern State University

CMPS 2143: Object Oriented Programming

Course Notes

Professor: Griffin

Fall 2020

Last Updated: January 13, 2021

Contents

1 Introduction

1.1 Structured Programming

Lots of this section is taken from the wisdom of http://www.fredosaurus.com/

For many years (roughly 1970 to 1990), structured programming was the most common way to organize a program and it is characterized by a functional-decomposition style. This simply means you break down your programs (the algorithms in which you are implementing) into small individual functions, such that each function handles a very specific task. Then functions that dealt with similar tasks (e.g. math, or input/output) could be grouped together in a cohesive library which then could be re-used at a later time (code re-use). This did allow programmers at the time to get more organized. But it didn't solve every issue.

It was hard to write a set of functions that were robust and could easily be used to solve other problems as well. When functions (or any programming constructs) are highly interdependent with each other, they are referred to as tightly coupled. This is not a good thing. The goal is to write code that is high in cohesion and low in coupling. Programmers at the time were finding it hard to achieve this goal when projects started getting bigger and bigger increasing the complexity of the problems. They needed better tools.

1.2 Object-Oriented Programming (OOP)

This is just an overview, so I won't delve too deep into any one topic. We will cover all of these things in depth throughout the semester.

Object-Oriented Programming groups related data and functions together in a *class*, generally making data private and only some functions public. Restricting access decreases coupling and increases cohesion. While it is not a panacea, it has proven to be very effective in reducing the complexity increase with large programs. For small programs may be difficult to see the advantage of OOP over structured programming because there is little complexity regardless of how it's written. Many of the mechanics of OOP are easy to demonstrate, but it is somewhat harder to create small, convincing examples.

OOP is often said to incorporate three techniques: inheritance , encapsulation , and polymorphism . Of these, you should first devote yourself to choosing the right classes (possibly difficult) and getting the encapsulation right (fairly easy). Inheritance and polymorphism are not even present in many programs, so you can ignore them at that start.

1.3 Inheritance

Inheritance means that a new class can be defined in terms of an existing class. Hmmm. What does that mean? Think of a super hero using "power absorption" to gain the power of another hero. This is how I picture inheritance, absorbing the power of another class! When you inherit

from another class (depending on a few things to be covered later) you gain access to their methods and data! Just like absorbing their power.

There are three common terminologies for the new class that "absorbed" or "inherited" new powers: the *derived* class, the *child* class, or the *subclass*. The original class is the *base* class, the *parent* class, or the *superclass*. The new child class inherits all capabilities of the parent class and then adds its own fields and methods. Although inheritance is very important, especially in many libraries, it is often not used in every day application. There are other techniques (like composition) that tend to fit better in our problem solving world.

1.4 Encapsulation

This is the grouping data and functions together and keeping their implementation details private. By restricting access to private functions and data we can also reduce *coupling*, which increases the ability to create large programs. In other words encapsulation helps us create entities that are highly self sufficient. They have their own methods (functions) and their own data, so they don't necessarily have to depend on lots of other entities to work (low coupling).

Classes also encourage *coherence*, which means that a given class does one thing. By increasing coherence, a program becomes easier to understand, more simply organized, and this better organization is reflected in a further reduction in coupling.

1.5 Polymorphism

Polymorphism is the ability of different functions to be invoked with the same name. There are two forms.

1.5.1 Static polymorphism

This is the common case of *overriding* a function by providing additional definitions with different numbers or types of parameters. The compiler matches the parameter list to the appropriate function.

1.5.2 Dynamic polymorphism

This is much different and relies on parent classes to define *virtual functions* which child classes may redefine. When this virtual member function is called for an object of the parent class, the execution dynamically chooses the appropriate function to call - the parent function if the object really is the parent type, or the child function if the object really is the child type. This explanation is too brief to be useful without an example, but that will have to be written latter.
Again. Just in introduction to get some terms rolling around in your head. We will re-visit all these concepts throughout the entire semester.
Source: http://www.fredosaurus.com/

2 Pointers

In C++, pointers are variables that store the memory addresses of other variables.

2.1 Addresses in C++

If we have a variable `var` in our program, `&var` will give us its address in the memory.

For example,

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

int main()
{
    // declare variables
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;

    // print address of var1
    cout << "Address of var1: "<< &var1 << endl;

    // print address of var2
    cout << "Address of var2: " << &var2 << endl;

    // print address of var3
    cout << "Address of var3: " << &var3 << endl;
}
```

Output

```
Address of var1: 0x7fff5fbff8ac
Address of var2: 0x7fff5fbff8a8
Address of var3: 0x7fff5fbff8a4
```

Here, `0x` at the beginning represents the address is in the hexadecimal form. Notice that the first address differs from the second by 4 bytes and the second address differs from the third by 4 bytes. This is because the size of an `int` variable is 4 bytes in a 64-bit system.

Note: You may not get the same results when you run the program.

2.2 C++ Pointers

As mentioned above, pointers are used to store addresses rather than values.
Here is how we can declare pointers.

```
int *pointVar;
```

Here, we have declared a pointer `pointVar` of the `int` type. We can also declare pointers in the following way.

```
int* pointVar; // preferred syntax
```

Let's take another example of declaring pointers.

```
int* pointVar, p;
```

Here, we have declared a pointer `pointVar` and a normal variable `p`.

Note: The $*$ operator is used after the data type to declare pointers.

2.2.1 Assigning Addresses to Pointers

Here is how we can assign addresses to pointers:

```
int* pointVar, var;
var = 5;

// assign address of var to pointVar pointer
pointVar = &var;
```

Here, `5` is assigned to the variable `var`. And, the address of `var` is assigned to the `pointVar` pointer with the code `pointVar = &var`.

2.2.2 Get the Value from the Address Using Pointers

To get the value pointed by a pointer, we use the $*$ operator. For example:

```
int* pointVar, var;
var = 5;

// assign address of var to pointVar
pointVar = &var;

// access value pointed by pointVar
cout << *pointVar << endl;   // Output: 5
```

In the above code, the address of var is assigned to `pointVar`. We have used the `*pointVar` to get the value stored in that address.
When $*$ is used with pointers, it's called the **dereference operator**. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, `*pointVar = var`.

Note: In C++, `pointVar` and `*pointVar` is completely different. We cannot do something like `*pointVar = &var;`

2.2.3 Example 2: Working of C++ Pointers

```cpp
#include <iostream>
using namespace std;
int main() {
    int var = 5;

    // declare pointer variable
    int* pointVar;

    // store address of var
    pointVar = &var;

    // print value of var
    cout << "var = " << var << endl;

    // print address of var
    cout << "Address of var (&var) = " << &var << endl
         << endl;

    // print pointer pointVar
    cout << "pointVar = " << pointVar << endl;

    // print the content of the address pointVar points to
    cout << "Content of the address pointed to by pointVar (*pointVar) = " << *pointVar << endl;

    return 0;
}
```

Output:

```
var = 5
Address of var (&var) = 0x61ff08

pointVar = 0x61ff08
Content of the address pointed to by pointVar (*pointVar) = 5
```

2.2.4 Changing Value Pointed by Pointers

If `pointVar` points to the address of `var`, we can change the value of `var` by using `*pointVar`.
For example,

Figure 1: Working of C++ pointers

```
int var = 5;
int* pointVar;

// assign address of var
pointVar = &var;

// change value at address pointVar
*pointVar = 1;

cout << var << endl; // Output: 1
```

Here, `pointVar` and `&var` have the same address, the value of `var` will also be changed when `*pointVar` is changed.

2.2.5 Example 3: Changing Value Pointed by Pointers

```
#include <iostream>
using namespace std;
int main() {
    int var = 5;
    int* pointVar;

    // store address of var
    pointVar = &var;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
```

```
    cout << "*pointVar = " << *pointVar << endl
         << endl;

    cout << "Changing value of var to 7:" << endl;

    // change value of var to 7
    var = 7;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl
         << endl;

    cout << "Changing value of *pointVar to 16:" << endl;

    // change value of var to 16
    *pointVar = 16;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl;
    return 0;
}
```

Output

```
var = 5
*pointVar = 5

Changing value of var to 7:
var = 7
*pointVar = 7

Changing value of *pointVar to 16:
var = 16
*pointVar = 16
```

2.3 Common mistakes when working with pointers

Suppose, we want a pointer `varPoint` to point to the address of `var`. Then,

```
int var, *varPoint;

```

```
3  // Wrong!
4  // varPoint is an address but var is not
5  varPoint = var;
6
7  // Wrong!
8  // &var is an address
9  // *varPoint is the value stored in &var
10 *varPoint = &var;
11
12 // Correct!
13 // varPoint is an address and so is &var
14 varPoint = &var;
15
16  // Correct!
17 // both *varPoint and var are values
18 *varPoint = var;
```

3 Arrays and Lists

Data structures (1063) is the prerequisite for this course. Some of you have had Algorithms (3013) already but I need to make sure you have your basics down. Data structures (IMO) should stress the following concepts:

- Arrays
- Lists (Using Pointers not using STL)
- Array based structures
- List based structures

If you struggle with any of these concepts you may want to come see me, or review the following materials:

1. Arrays
2. Pointers and Memory
3. Linked List Basics

3.1 Array vs List

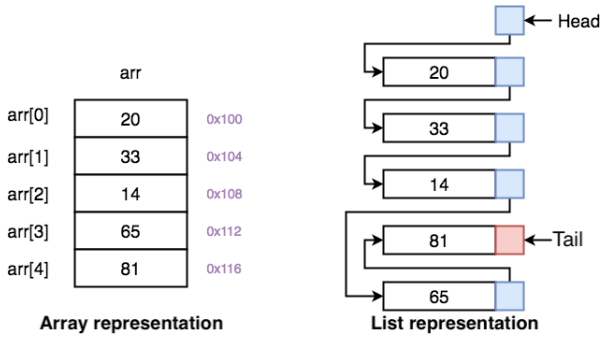
Since were talking about arrays and lists, lets break it down and summarize a little bit. When creating a data structure you need to ask yourself: "Do I use an array to store my data? Or do I used a linked list?" In computer science we use the terms: **array based structure** and **list based structure** for our two choices on how to store data in memory. However, to create list based structures you need to "link" memory locations, and so you were introduced to **pointers** which allowed you to create **linked lists**. Both arrays and linked lists have pros and cons associated with them, and understanding the pros and cons of each structure will help you understand why certain algorithms work better when implemented using one or the other. This is not an algorithms course, but understanding the differences between arrays and linked structures is pretty fundamental to computer science, so I go over them whenever I can. Remember, a linked structure can do anything an array can do (and vice versa) it just may not do it very well. Again, this course is OOP and were not designing algorithms, but coding is coding and you should be comfortable with linked lists and array. Below is a summary of both:

Array:

1. Direct access (random access) using subscript.
2. Growing and shrinking is costly.
3. Bad for lots of insertions and deletions (because of empty slots and need for shifting).
4. Good for searching (if items are ordered).

Linked Lists:

1. Access requires traversal using a pointer.
2. Grows and shrinks easily.
3. Good for lots of insertions and deletions (because of previous point).
4. Bad for searching (linear only).

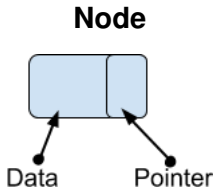


Comparison	Array	Linked List
Generic	Contiguous memory locations of fixed size.	Dynamically allocated memory locations linked by pointers.
Size	Fixed at allocation.	Grows and shrinks easily.
Order of the elements	Stored consecutively	Stored randomly
Accessing	Direct access using subscript.	Sequentially accessed from head or other pointer.
Insertion and deletion	Slow if shifting is required.	Easier, fast and efficient.
Searching	Binary search (if ordered values) and linear search.	Linear search only.
Memory required	Size of data stored.	Size of data stored + pointer.

Table 1: Array VS Linked List

3.2 Linked List Refresher

A linked list is a sequence of data structures, which are connected together via links (pointers).



Linked List is a sequence of nodes which contains items. Each node contains a connection to another node. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Node**: Each node of a linked list stores a data element. Something simple like an **int** or complex like a class or **struct**.

- **Next**: Each node of a linked list contains a **pointer** to the next node. Typically called **Next** .
- **Head**: Every linked list needs a way to start accessing it. We typically call this pointer **Head** or **Front** .

3.2.1 Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

Looking at the image we can see:

- A *Head* pointer that gives us access to the list. Lose *Head*, and lose the list!
- Each Node has data, and a pointer to the next node in the list..
- The last Node's "Next" pointer, points to *NULL*. This tells us its the end of the list.

3.2.2 Types of Linked List

Following are the various types of linked list.

- **Singly Linked List** Only traverse this list one way.
- **Doubly Linked List** Can traverse this list forwards and backwards.
- **Circular Linked List** Last item contains link of the first element as next and the first element has a link to the last element as previous.

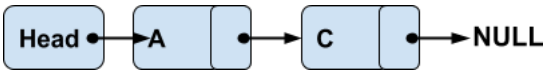
3.2.3 Basic Operations

Following are the basic operations supported by a list.

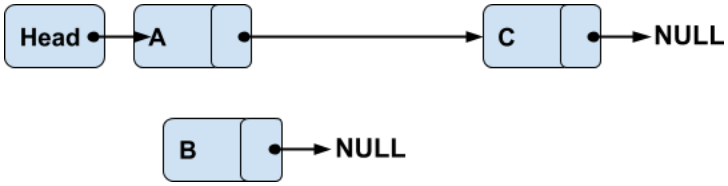
- **Insert** Adds an element at the beginning or end of the list, depending on your implementation.
- **Search** Searches an element using the given key.
- **Delete** Deletes an element from the list. If its not the first or last Node, then we accompany this with a search to find the proper node to delete using the given key.
- **Print** Prints the list. Mostly for us programmers to debug our linked list code and make sure its working as expected.

3.2.4 Insertion Operation

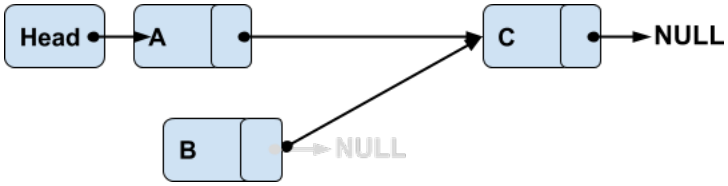
Adding a new node to a list requires the creation of a new node, and the manipulation of a couple of pointers. Lets add a node to the following list:



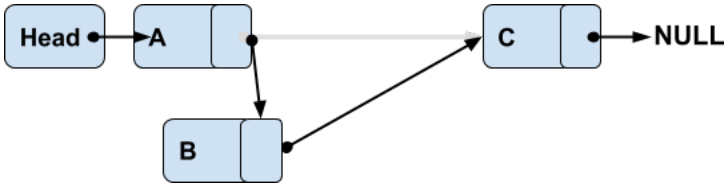
We start by creating a new node "**B**" which we want to add between **A** and **C** (for visual effect I moved **C** further away):



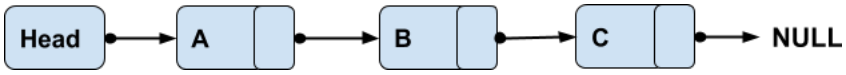
Our first step is to "link" the new node into the list, by pointing its **next** pointer to **C**.



Then we complete the insertion by pointing **A** to **B**



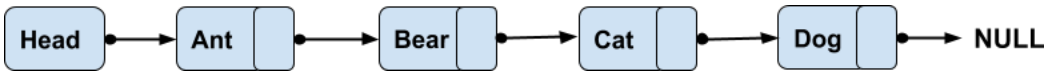
Your resulting linked list:



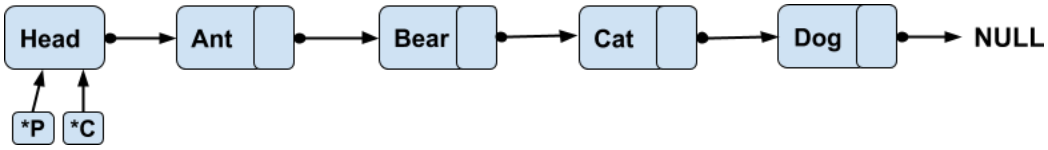
Its similar if you insert a node at the beginning or end of a list. This is just an overview, and I won't be posting any code specific examples. https://repl.it/@rugbyprof/intlinkedlistmain.cpp is a linked list implementation on Repl.it.

3.2.5 Deletion Operation

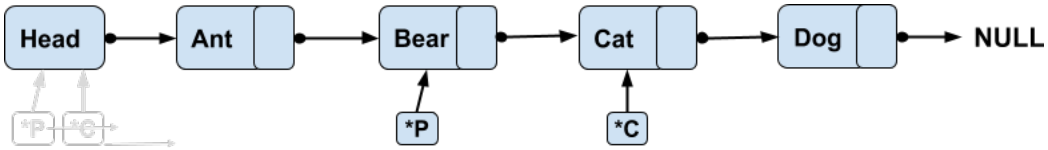
Given the list below, lets say we want to delete the node with the value "Cat" in it.



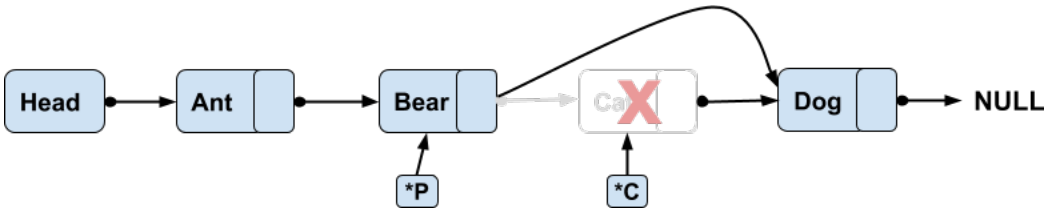
Deletion is also a more than one step process. You first need to traverse the list to find the item you want to delete, and this starts by placing two pointers at the head of our list. We will be using "P" for previous, and "C" for current.



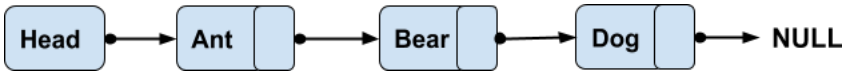
We then traverse the list searching to find the "key" we want to delete. We use the "C" pointer to look at the data in each node, and we use the "P" pointer to trail behind staying one node behind the current pointer.



Why do we need two pointers? Look at the image above. Does the "C" pointer have the ability to make the previous node *bypass* the node it is pointing to? NO! That's why we use a "P" pointer. It gives us the ability to jump over "current" and thereby removing the node from our list.



Resulting list after "Cat" is deleted.



3.3 Stacks and Queues

Something else you should have taken away from your previous CS course is the concepts of two simple data structures: 1) **Stacks** and 2) **Queues**. Not only the concept of how they work (LIFO v FIFO), but also how to implement either of them using an array or a list.

1. Stacks and Queues Helper Material

There are many other things you should have taken away from your data structures course, but to continue in this class with some comfort, you really should comfortably grasp the following. Each of the items is a link to a gist with example code.

1. Array Based Stack
2. List Based Stack
3. Array Based Queue
4. List Based Queue

If you can comfortably write all of those combinations basically from scratch, you are in a good position to continue. If not, practice!!

4 Scope

Scope deals with where variables are "visible" in a program. There are many nuances when talking about scope so if you are looking for an in depth explanation on all the types of scope try this site: cppreference.com. This section on scope is just a simple refresher of the most basic scoping rules.

What we usually deal with in class is `block scope` or in other words `global` and `local` variables. Like the term implies (to me anyway), `block scope` classifies variables declared within a block as `local` to that block. Variables declared outside of any block are classified as `global` variables. But as always, there are some nuances that are best explained using examples.

4.1 Block

First, what is a `block` ? A block is a pair of curly braces `{}` used as an organizational construct to define the body of a function, class, method, loop, etc. but it also can be used outside of those constructs to simply define `scope` . The example below multiples blocks nested within each other and not a function, loop, or if statement in site. Would this compile? Sure, if you added a few things. But it by itself is valid (OK there's one error, can you find it?). The question is, whats going with those nested blocks?

```
{
    int x = 1;
    cout << x << endl;
    {
        cout << x << endl;
        int x = 2;
        int y = 5;
        cout << x << endl;
        {
            cout << x << endl;
            int x = 3;
            cout << x << endl;
        }
        cout << x << endl;
    }
    cout << x << endl;
    cout << y << endl;
}
```

Block Scope Rules:

- Blocks are portions of code contained within curly braces `{....}` .
- Identifiers declared within a block have block scope and are visible from their points of definition to the end of the innermost containing block.
- A duplicate identifier name in a block hides the value of an identifier with the same name defined outside the block.

- A variable name declared in a block is local to that block. It can be used only in it and the other blocks contained under it.

Example 1:

- This example has 3 code blocks, or 3 scope levels
- There is an `x` declared at every scope level, so each `cout` should reference the `x` at its own level, **if the `x` was declared before it was used.**
- If there is no variable declared within a code block before it is used, then it will use the value from the next level up.
- If there is no variable declared in the previous level, it will continue to go up levels until it finds a variable with the same name, or error if one doesn't exist.

```
{ First Level
    int x = 1;
    cout << x << endl;
    { Second Level
        cout << x << endl;
        int x = 2;
        int y = 5;
        cout << x << endl;
        { Third Level
            cout << x << endl;
            int x = 3;
            cout << x << endl;
        }
        cout << x << endl;
    }
    cout << x << endl;
    cout << y << endl;
}
```

```
1  {
2      int x = 1;
3      cout << x << endl;          // OUTPUTS: 1 (uses level 1 x)
4      {
5          cout << x << endl;      // OUTPUTS: 1 (uses level 1 x)
6          int x = 2;
7          int y = 5;
8          cout << x << endl;      // OUTPUTS: 2 (uses level 2 x)
9          {
10             cout << x << endl;  // OUTPUTS: 2 (uses level 2 x)
11             int x = 3;
12             cout << x << endl;  // OUTPUTS: 3 (uses level 3 x)
13         }
14         cout << x << endl;      // OUTPUTS: 2 (uses level 2 x)
15     }
16     cout << x << endl;          // OUTPUTS: 1 (uses level 1 x)
```

```
    cout << y << endl;          // ERROR
}
```

Example 2:

```
{
    int x = 1;
    int y = 2;
    cout << x << endl;          // OUTPUTS: 1 (uses level 1 x)
    {
        cout << x << endl;      // OUTPUTS: 1 (uses level 1 x)
        {
            int y = 4;
            cout << x << endl;  // OUTPUTS: 1 (uses level 1 x)
            cout << y << endl;  // OUTPUTS: 4 (uses level 3 y)
        }
        cout << x << endl;      // OUTPUTS: 1 (uses level 1 x)
    }
    cout << x << endl;          // OUTPUTS: 1 (uses level 1 x)
    cout << y << endl;          // OUTPUTS: 2 (uses level 1 y)
}
```

4.2 Local Variables

Below we have a variable declared in the function `someAge`

```
/**
 * The variable age is local to this function.
 */
void someAge() {
    int age=18;
}

/**
 * The variable age cannot be accessed from main
 */
int main()
{
    cout<<"Age is: "<<age;
    return 0;
}
```

Output:

```
Error: age was not declared in this scope
```

To get access to that local variable declared within that functions code block, you need to return it.

```
/**
 * The variable age is local to this function.
 */
int someAge() {
    int age=18;

    return age;
}

/**
 * Were still not accessing the variable from someAge()
 * we just got a copy of it.
 */
int main()
{
    cout<<"Age is: "<<someAge();
    return 0;
}
```

Output:

```
Age is: 18
```

4.3 Global Variables

- Global variables are defined outside of all the functions, usually at the top of your program.
- They will hold their value throughout the life-time of your program.
- They can also be accessed by any function, unless a variable with the same name is declared within the functions code block.

```
// Global variable declaration:
int g;

int main () {
   // Local variable declaration:
   int a, b;

   // actual initialization
   a = 10;
   b = 20;
   g = a + b;

   cout << g;

   return 0;
}
```

Output:

```
30
```

```
// Global variable declaration:
int g = 20;

int main () {
   // Local variable declaration:
   int g = 10;

   cout << g;

   return 0;
}
```

Output:

```
10
```

5 Classes and Objects

Source: ://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.
A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

5.1 C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.
A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword class as follows

```
class Box {
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
```

The keyword public determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in a sub-section.

5.2 Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

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

5.3 Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear

```
#include <iostream>

using namespace std;

class Box {
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};

int main() {
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;

   // volume of box 1
   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

6 Access Control

This section covers using the **public**, **protected** and **private** keywords to specify levels of access when one class inherits from another. Using inheritance, we can derive a child class from the base class in different access modes. There are many terms you may hear or when discussing inheritance. Here are a few different terms when referring to the class that is inherited from vs the class that does the inheriting.

Inherited From	Does Inheriting
Super Class	Sub Class
Parent Class	Child Class
Base Class	Derived Class

For example,

```
class Base {
    Base(){}
};

class Derived : public Base {
    Derived(){}
};
```

Notice the keyword **public** in the declaration of *class derived*. This means that we have created a derived class from the base class in **public mode**. Alternatively, we can also derive classes in **protected** or **private** modes.

These 3 keywords (public, protected, and private) are known as **access specifiers** when a sub class inherits from a base class.

6.1 Three Access Methods: public, protected and private

public, **protected**, and **private** inheritance have the following features:

- **public** inheritance makes *public* members of the base class *public* in the derived class, and the *protected* members of the base class remain *protected* in the derived class.
- **protected** inheritance makes the *public* and *protected* members of the base class *protected* in the derived class.
- **private** inheritance makes the *public* and *protected* members of the base class *private* in the derived class.

Note: private members of the base class are inaccessible to the derived class.

6.2 General Access Example

Below is an example using all three access methods:

```
class Base {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

class PublicDerived: public Base {
    // x is public
    // y is protected
    // z is not accessible from PublicDerived
};

class ProtectedDerived: protected Base {
    // x is protected
    // y is protected
    // z is not accessible from ProtectedDerived
};

class PrivateDerived: private Base {
    // x is private
    // y is private
    // z is not accessible from PrivateDerived
}
```

6.3 Accessibility in Public Inheritance

Below is an example where a child class inherits from the base class with a **public** access level:

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

class Base {
   private:
    int pvt = 1;

   protected:
    int prot = 2;

   public:
    int pub = 3;
```

```
    // function to access private member
    int getPVT() {
        return pvt;
    }
};

class PublicDerived : public Base {
   public:
    // function to access protected member from Base
    int getProt() {
        return prot;
    }
};

int main() {
    PublicDerived object1;
    cout << "Private = " << object1.getPVT() << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.pub << endl;
    return 0;
}
```

Output

```
Private = 1
Protected = 2
Public = 3
```

Here, we have derived `PublicDerived` from `Base` in **public mode**.

As a result, in `PublicDerived` :

- `prot` is inherited as protected.
- `pub` and `getPVT()` are inherited as public.
- `pvt` is inaccessible since it is private in Base.

Since **private** and **protected** members are not accessible, we need to create public functions `getPVT()` and `getProt()` to access them:

```
// Error: member "Base::pvt" is inaccessible
cout << "Private = " << object1.pvt;

// Error: member "Base::prot" is inaccessible
cout << "Protected = " << object1.prot;
```

6.3.1 Public Inheritance Overview

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

6.4 Accessibility in Protected Inheritance

Below is an example where a child class inherits from the base class with a `protected` access level:

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

class Base {
   private:
    int pvt = 1;

   protected:
    int prot = 2;

   public:
    int pub = 3;

    // function to access private member
    int getPVT() {
        return pvt;
    }
};

class ProtectedDerived : protected Base {
   public:
    // function to access protected member from Base
    int getProt() {
        return prot;
    }

    // function to access public member from Base
    int getPub() {
        return pub;
    }
};

int main() {
    ProtectedDerived object1;
    cout << "Private cannot be accessed." << endl;
    cout << "Protected = " << object1.getProt() << endl;
```

```
    cout << "Public = " << object1.getPub() << endl;
    return 0;
}
```

Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived `ProtectedDerived` from `Base` in **protected mode**.

As a result, in `ProtectedDerived` :

- `prot` , `pub` and `getPVT()` are inherited as **protected**.
- `pvt` is inaccessible since it is **private** in `Base` .

As we know, **protected** members cannot be accessed directly.

As a result, we cannot use `getPVT()` from `ProtectedDerived` . That is also why we need to create the `getPub()` function in `ProtectedDerived` in order to access the `pub` variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

6.4.1 Protected Inheritance Overview

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes (inherited as protected variables)

6.5 Accessibility in Private Inheritance

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

class Base {
   private:
    int pvt = 1;

   protected:
    int prot = 2;
```

```
   public:
    int pub = 3;

    // function to access private member
    int getPVT() {
        return pvt;
    }
};

class PrivateDerived : private Base {
   public:
    // function to access protected member from Base
    int getProt() {
        return prot;
    }

    // function to access private member
    int getPub() {
        return pub;
    }
};

int main() {
    PrivateDerived object1;
    cout << "Private cannot be accessed." << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.getPub() << endl;
    return 0;
}
```

Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived `PrivateDerived` from `Base` in private mode.

As a result, in `PrivateDerived` :

- `prot` , `pub` and `getPVT()` are inherited as **private**.
- `pvt` is inaccessible since it is **private** in `Base` .

As we know, private members cannot be accessed directly.

As a result, we cannot use `getPVT()` from `PrivateDerived` . That is also why we need to create the `getPub()` function in `PrivateDerived` in order to access the `pub` variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

6.5.1 Private Inheritance Overview

Accessibility	**private members**	**protected members**	**public members**
Base Class	Yes	Yes	Yes
Derived Class	No	Yes *(inherited as private variables)*	Yes *(inherited as protected variables)*

7 C++ Operator Overloading

One of the nice features of C++ is that you can give special meanings to operators, when they are used with user-defined classes. This is called operator overloading. You can implement C++ operator overloads by providing special member-functions on your classes that follow a particular naming convention. For example, to overload the `+` operator for your class, you would provide a member-function named `operator+` on your class.

The following set of operators is commonly overloaded for user-defined classes:

$=$	assignment operator
$+, -, *$	binary arithmetic operators
+= -= *=	compound assignment operators
== !=	comparison operators

7.1 Guidelines

Here are some guidelines for implementing these operators. These guidelines are very important to follow, so definitely get in the habit early.

7.1.1 Assignment Operator =

The assignment operator has a signature like this:

```
class MyClass {
public:
...
   MyClass & operator=(const MyClass &rhs);
...
}

MyClass a, b;
...
b = a;   // Same as b.operator=(a);
```

Notice that the `=` operator takes a `const` reference to the right hand side of the assignment. The reason for this should be obvious, since we don't want to change that value; we only want to change what's on the left hand side.

Also, you will notice that a reference is returned by the assignment operator. This is to allow operator chaining. You typically see it with primitive types, like this:

```
int a, b, c, d, e;

a = b = c = d = e = 42;
```

This is interpreted by the compiler as:

```
a = (b = (c = (d = (e = 42))));
```

In other words, assignment is right-associative. The last assignment operation is evaluated first, and is propagated leftward through the series of assignments. Specifically:

- `e = 42` assigns `42` to `e`, then returns `e` as the result
- The value of `e` is then assigned to `d`, and then `d` is returned as the result
- The value of `d` is then assigned to `c`, and then `c` is returned as the result
- and so on ...

Now, in order to support operator chaining, the assignment operator must return some value. The value that should be returned is a reference to the left-hand side of the assignment.

Notice that the returned reference is not declared `const`. This can be a bit confusing, because it allows you to write crazy stuff like this:

```
MyClass a, b, c;
...
(a = b) = c;  // What??
```

At first glance, you might want to prevent situations like this, by having `operator=` return a **const** reference. However, statements like this will work with primitive types. And, even worse, some tools actually rely on this behavior. Therefore, it is important to return a `non-const` reference from your `operator=`. The rule of thumb is, "If it's good enough for `int's`, it's good enough for user-defined data-types."

So, for the hypothetical MyClass assignment operator, you would do something like this:

```
// Take a const-reference to the right-hand side of the assignment.
// Return a non-const reference to the left-hand side.
MyClass& MyClass::operator=(const MyClass &rhs) {
...  // Do the assignment operation!

return *this;  // Return a reference to myself.
}
```

Remember, this is a pointer to the object that the member function is being called on. Since `a = b` is treated as `a.operator=(b)`, you can see why it makes sense to return the object that the function is called on; object `a` is the left-hand side.
But, the member function needs to return a reference to the object, not a pointer to the object. So, it returns `*this`, which returns what this points at (i.e. the object), not the pointer itself. (In C++, instances are turned into references, and vice versa, pretty much automatically, so even though `*this` is an instance, C++ implicitly converts it into a reference to the instance.)

Now, one more very important point about the assignment operator:

YOU MUST CHECK FOR SELF-ASSIGNMENT!

This is especially important when your class does its own memory allocation. Here is why: The typical sequence of operations within an assignment operator is usually something like this:

```
MyClass& MyClass::operator=(const MyClass &rhs) {
// 1.  Deallocate any memory that MyClass is using internally
// 2.  Allocate some memory to hold the contents of rhs
// 3.  Copy the values from rhs into this instance
// 4.  Return *this
}
```

Now, what happens when you do something like this:

```
MyClass mc;
...
mc = mc;     // BLAMMO.
```

You can hopefully see that this would wreak havoc on your program. Because `mc` is on the left-hand side and on the right-hand side, the first thing that happens is that `mc` releases any memory it holds internally. But, this is where the values were going to be copied from, since `mc` is also on the right-hand side! So, you can see that this completely messes up the rest of the assignment operator's internals.

The easy way to avoid this is to **CHECK FOR SELF-ASSIGNMENT**. There are many ways to answer the question, "Are these two instances the same?" But, for our purposes, just compare the two objects' `addresses` . If they are the same, then don't do assignment. If they are different, then do the assignment.

So, the correct and safe version of the MyClass assignment operator would be this:

```
MyClass& MyClass::operator=(const MyClass &rhs) {
    // Check for self-assignment!
    if (this == &rhs)      // Same object?
        return *this;        // Yes, so skip assignment, and just return *this.

    ... // Deallocate, allocate new space, copy values...

    return *this;
}
```

Or, you can simplify this a bit by doing:

```
MyClass& MyClass::operator=(const MyClass &rhs) {

    // Only do assignment if RHS is a different object from this.
    if (this != &rhs) {
      ... // Deallocate, allocate new space, copy values...
    }

    return *this;
}
```

Remember that in the comparison, this is a pointer to the object being called, and `rhs` is a pointer to the object being passed in as the argument. So, you can see that we avoid the dangers of self-assignment with this check.
In summary, the guidelines for the assignment operator are:

- Take a const-reference for the argument (the right-hand side of the assignment).
- Return a reference to the left-hand side, to support safe and reasonable operator chaining. (Do this by returning `*this` .)
- Check for self-assignment, by comparing the pointers (this to `rhs`).

7.1.2 Compound Assignment Operators +=, -=, *=

I discuss these before the arithmetic operators for a very specific reason, but we will get to that in a moment. The important point is that these are destructive operators, because they update or replace the values on the left-hand side of the assignment. So, you write:

```
MyClass a, b;
...
a += b;    // Same as a.operator+=(b)
```

In this case, the values within a are modified by the `+= operator` .
How those values are modified isn't very important - obviously, what `MyClass` represents will dictate what these operators mean.
The member function signature for such an operator should be like this:

```
MyClass & MyClass::operator+=(const MyClass &rhs) {
    ...
}
```

We have already covered the reason why `rhs` is a `const-reference` . And, the implementation of such an operation should also be straightforward. But, you will notice that the operator returns a `MyClass-reference` , and a `non-const` one at that. This is so you can do things like this:

```
MyClass mc;
...
(mc += 5) += 3;
```

Don't ask me why somebody would want to do this, but just like the normal assignment operator, this is allowed by the primitive data types. Our `user-defined` data types should match the same general characteristics of the `primitive` data types when it comes to operators, to make sure that everything works as expected.

This is very straightforward to do. Just write your compound assignment operator implementation, and return `*this` at the end, just like for the regular assignment operator. So, you would end up with something like this:

```
MyClass & MyClass::operator+=(const MyClass &rhs) {
    ...   // Do the compound assignment work.

    return *this;
}
```

As one last note, in general you should beware of self-assignment with compound assignment operators as well.

7.1.3 Binary Arithmetic Operators +, -, *

The binary arithmetic operators are interesting because they don't modify either operand - they actually return a new value from the two arguments. You might think this is going to be an annoying bit of extra work, but here is the secret:

`Define your binary arithmetic operators using your compound assignment operators.`

So, you have already implemented your `+= operator`, and now you want to implement the `+ operator`. The function signature should be like this:

```
// Add this instance's value to other, and return a new instance
// with the result.
const MyClass MyClass::operator+(const MyClass &other) const {
    MyClass result = *this;     // Make a copy of myself.  Same as MyClass result(*this);
    result += other;            // Use += to add other to the copy.
    return result;              // All done!
}
```

Actually, this explicitly spells out all of the steps, and if you want, you can combine them all into a single statement, like so:

```
// Add this instance's value to other, and return a new instance
// with the result.
const MyClass MyClass::operator+(const MyClass &other) const {
    return MyClass(*this) += other;
}
```

This creates an unnamed instance of `MyClass`, which is a copy of `*this`. Then, the `+= operator` is called on the temporary value, and then returns it. If that last statement doesn't make sense to you yet, then stick with the other way, which spells out all of the steps. But, if you understand exactly what is going on, then you can use that approach.
You will notice that the `+ operator` returns a `const instance`, not a const reference. This is so that people can't write strange statements like this:

```
MyClass a, b, c;
...
(a + b) = c;    // Wuh...?
```

This statement would basically do nothing, but if the `+ operator` returns a `non-const` value, it will compile! So, we want to return a `const instance`, so that such madness will not even be allowed to compile.

To summarize, the guidelines for the binary arithmetic operators are:

- Implement the compound assignment operators from scratch, and then define the binary arithmetic operators in terms of the corresponding compound assignment operators.

- Return a const instance, to prevent worthless and confusing assignment operations that shouldn't be allowed.

7.1.4 Comparison Operators == and !=

The comparison operators are very simple. Define `==` first, using a function signature like this:

```
bool MyClass::operator==(const MyClass &other) const {
    ...  // Compare the values, and return a bool result.
}
```

The internals are very obvious and straightforward, and the `bool return-value` is also very obvious. The important point here is that the `!= operator` can also be defined in terms of the `== operator`, and you should do this to save effort. You can do something like this:

```
bool MyClass::operator!=(const MyClass &other) const {
    return !(*this == other);
}
```

That way you get to reuse the hard work you did on implementing your `== operator`. Also, your code is far less likely to exhibit inconsistencies between `==` and `!=`, since one is implemented in terms of the other.

8 Inheritance

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class). In an extreme oversimplification lets call the some classes "givers" and other classes "takers". A "giver" lets other classes use data members and methods they have already defined to help speed development. "Takers" use those methods and/or data members to specialize it in some way by adding their own little tweaks. Lets replace "giver" with "generic" and "taker" with "specialized". Now we have this idea of taking a "generic" class and "specializing" it by adding small tweaks. This is the core idea of Object Oriented Design.
There are more accepted terms for "giver"/"taker" or "generic"/"specialize" which I have summarized below. After you read the acceptable terms below ... never use my terms from above again when discussing OOP 😆

Inherited From	Does Inheriting
~~Giver~~	~~Taker~~
~~Generic~~	~~Specialized~~
Super Class	Sub Class
Parent Class	Child Class
Base Class	Derived Class

The derived class inherits the features from the base class and can have additional features of its own.

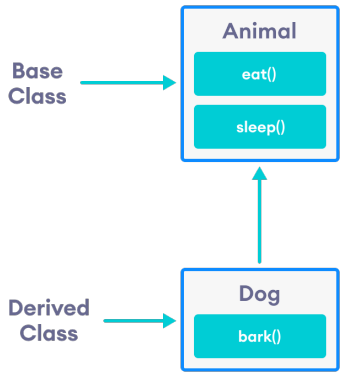


Figure 2: Inheritance

For example,

```
class Animal {
    // eat() function
    // sleep() function
};

class Dog : public Animal {
    // bark() function
};
```

Here, the `Dog` class is derived from the `Animal` class. Since `Dog` is derived from `Animal`, members of `Animal` are accessible to `Dog`.
Notice the use of the keyword `public` while inheriting Dog from Animal.

```
class Dog : public Animal {...};
```

We can also use the keywords `private` and `protected` instead of `public`. You can read more about the differences between using `private`, `public` and `protected` here.

8.1 IS-A relationship

Inheritance is an **is-a relationship**. We use inheritance only if the `derived class` **is-a** `base class.`

Or more specifically the `derived` class has an **is-a relationship** with the `base` class.

Here are some examples:

- A car **is a** vehicle.
- Orange **is a** fruit.
- A surgeon **is a** doctor.
- A dog **is an** animal.
- A wheel ~~**is a** car~~.
- A card ~~**is a** deck~~.

8.2 Simple Example of Inheritance

```
// C++ program to demonstrate public members
#include <iostream>
using namespace std;

// base class
class Animal {

   public:
    void eat() {
        cout << "I can eat!" << endl;
    }
```

```
    void sleep() {
        cout << "I can sleep!" << endl;
    }
};

// derived class
class Dog : public Animal {

   public:
    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();

    // Calling member of the derived class
    dog1.bark();

    return 0;
}
```

Output

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

Here, `dog1` (the object of derived class `Dog`) can access members of the base class `Animal`.

```
// Calling members of the Animal class
dog1.eat();
dog1.sleep();
```

It's because `Dog` is inherited from `Animal`.

8.3 Protected Members

The access modifier `protected` is especially relevant when it comes to C++ inheritance.

Like `private` members, `protected` members are inaccessible outside of the class. However, they can be accessed by **derived classes** and **friend classes** / **functions**.

We need `protected` members if we want to hide the data of a class, but still want that data to be inherited by its derived classes.

8.3.1 Protected Members

```cpp
// C++ program to demonstrate protected members

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

// base class
class Animal {

   private:
    string color;

   protected:
    string type;

   public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }

    void setColor(string clr) {
        color = clr;
    }

    string getColor() {
        return color;
    }
};

// derived class
class Dog : public Animal {

   public:
```

```
    void setType(string tp) {
        type = tp;
    }

    void displayInfo(string c) {
        cout << "I am a " << type << endl;
        cout << "My color is " << c << endl;
    }

    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();
    dog1.setColor("black");

    // Calling member of the derived class
    dog1.bark();
    dog1.setType("mammal");

    // Using getColor() of dog1 as argument
    // getColor() returns string data
    dog1.displayInfo(dog1.getColor());

    return 0;
}
```

Output

```
I can eat!
I can sleep!
I can bark! Woof woof!!
I am a mammal
My color is black
```

Here, the variable `type` is `protected` and is thus accessible from the derived class `Dog`. We can see this as we have initialized `type` in the `Dog` class using the function `setType()`.

On the other hand, the `private` variable `color` cannot be initialized in `Dog`.

```
class Dog : public Animal {
```

```
    public:
      void setColor(string clr) {
          // Error: member "Animal::color" is inaccessible
          color = clr;
      }
};
```

Also, since the `protected` keyword hides data, we cannot access `type` directly from an object of `Dog` or `Animal` class.

```
// Error: member "Animal::type" is inaccessible
dog1.type = "mammal";
```

8.4 Access Modes in C++ Inheritance

We have learned about C++ access specifiers such as **public, private, and protected**. And so far, we have used the `public` keyword in order to inherit a class from a previously-existing base class. However, we can also use the `private` and `protected` keywords to inherit classes.

For example,

```
class Animal {
    // code
};

class Dog : private Animal {
    // code
};

class Cat : protected Animal {
    // code
};
```

The various ways we can derive classes are known as **access modes**. These access modes have the following effect:

1. **public:** If a derived class is declared in `public` mode, then the members of the base class are inherited by the derived class just as they are.
2. **private:** In this case, all the members of the base class become `private` members in the derived class.
3. **protected:** The `public` members of the base class become `protected` members in the derived class.

The `private` members of the base class are always `private` in the derived class.
See more about access control methods (Public,Protected,Private) here.
Function Overriding

---

9 Static Keyword

Static Keyword can be used with following,

- Static variable in functions
- Static Class Objects
- Static member Variable in class
- Static Methods in class

9.1 Static Variables inside Functions

- Static variables when used inside function are initialized only once, and then they hold there value even through function calls.
- These static variables are stored on static storage area , not in stack.

```
void counter(){
    static int count=0;
    cout << count++;
}

int main({
    for(int i=0;i<5;i++){
        counter();
    }
}
```

Output:

```
0 1 2 3 4
```

Let's see the same program's output without using static variable.

```
void counter(){
    int count=0;
    cout << count++;
}

int main({
    for(int i=0;i<5;i++){
        counter();
    }
}
```

Output:

```
0 0 0 0 0
```

- If we do not use `static` keyword, the variable `count` , is reinitialized every time when `counter()` function is called, and gets destroyed each time when `counter()` functions ends.
- But, if we make it `static` , once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.
- If you don't initialize a static variable, they are by default initialized to zero.

9.2 Static Class Objects

- Static keyword works in the same way for class objects too.
- Objects declared static are allocated storage in static storage area, and have scope till the end of program.
- Static objects are also initialized using constructors like other normal objects.
- Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc{
    int i;
    public:
    Abc(){
        i=0;
        cout << "constructor";
    }
    ~Abc(){
        cout << "destructor";
    }
};

void f(){
    cout<<"In f()"<<endl;
    static Abc obj;
    cout<<"Exiting f()"<<endl;
}

int main(){
    int x=0;
    if(x==0){
        f();
    }
    cout << "END";
}
```

Output:

```
Start main
Start f()
constructor
End f()
End main
destructor
```

You must be thinking, why was the `destructor` not called upon the end of the scope of if condition, where the reference of object `obj` should get destroyed. This is because object was `static` , which has scope till the program's lifetime, hence destructor for this object was called when main() function exits.

9.3 Static Data Member in Class

- `Static` data members of class are those members which are shared by all the objects.
- They are allocated a single piece of storage which is the same in every object. So, every object has its own copy of all data members **except static members** which they share with all other objects.
- `Static` member variables (data members) are not initialized using a constructor, because these are not dependent on object initialization.
- Also, `static members must be initialized explicitly` , always outside the class. If not initialized, Linker will give error.

```
class Foo
{
    public:
    static int i;
    Foo()
    {
        // construtor
        cout<<i<<endl;
    };
};

int Foo::i=1;

int main()
{
    Foo obj;
    cout << obj.i;   // prints value of i
}
```

Output:

```
1
```

Once the definition for static data member is made, user cannot redefine it. Though, arithmetic operations can be performed on it.

9.4 Static Member Functions

- These functions work for the class as whole rather than for a particular object of a class.
- It can be called using an object and the direct member access `.` operator.
- However, its more typical to call a static member function by itself, using class name and scope resolution `::` operator.
- Functions declared as static, cannot access ordinary data members and member functions, only other statically declared items within the class.

For example:
class Math private: static int curve; public: static int add(int a, int b) return a + b; static int sub(int a, int b) return a - b; static int mul(int a, int b) return a * b; ;
int main() cout«Math::add(2,3)«endl; cout«Math::sub(2,3)«endl; cout«Math::mul(2,3)«endl;
Output:

```
5
-1
6
```

10 Using Friend Functions

Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class. Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

```
class MyClass {
    private:
        int member1;
}

int main() {
    MyClass obj;

    // Error! Cannot access private members from here.
    obj.member1 = 5;
}
```

However, there is a feature in C++ called `friend functions` that break this rule and allow us to access member functions from outside the class. Similarly, there is a `friend class` as well, which we will discuss later.

10.1 Friend Function

A `friend function` can access the `private` and `protected` data of a class. We declare a friend function using the `friend` keyword inside the body of the class.

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

10.1.1 Example 1: Working of friend Function

```
// C++ program to demonstrate the working of friend function

#include <iostream>
using namespace std;

class Distance {
    private:
```

```
        int meter;

        // friend function
        friend int addFive(Distance);

    public:
        Distance() : meter(0) {}

};

// friend function definition
int addFive(Distance d) {

    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}

int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Output:

```
Distance: 5
```

Here, `addFive()` is a friend function that can access both `private` and `public` data members. Though this example gives us an idea about the concept of a friend function, it doesn't show any meaningful use. A more meaningful use would be operating on objects of two different classes. That's when the friend function can be very helpful.

10.1.2 Example 2: Add Members of Two Different Classes

```
// Add members of two different classes using friend functions

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {

    public:
        // constructor to initialize numA to 12
```

```
        ClassA() : numA(12) {}

    private:
        int numA;

         // friend function declaration
         friend int add(ClassA, ClassB);
};

class ClassB {

    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    private:
        int numB;

        // friend function declaration
        friend int add(ClassA, ClassB);
};

// access members of both classes
int add(ClassA objectA, ClassB objectB) {
    return (objectA.numA + objectB.numB);
}

int main() {
    ClassA objectA;
    ClassB objectB;
    cout << "Sum: " << add(objectA, objectB);
    return 0;
}
```

Output

```
Sum: 13
```

In this program, `ClassA` and `ClassB` have declared `add()` as a friend function. Thus, this function can access `private` data of both classes. One thing to notice here is the friend function inside `ClassA` is using the `ClassB` . However, we haven't defined `ClassB` at this point.

```
// inside classA
friend int add(ClassA, ClassB);
```

For this to work, we need a forward declaration of `ClassB` in our program.

```
// forward declaration
class ClassB;
```

10.2 friend Class in C++

We can also use a friend Class in C++ using the `friend` keyword. For example,

```
class ClassB;

class ClassA {
   // ClassB is a friend class of ClassA
   friend class ClassB;
   ... .. ...
}

class ClassB {
   ... .. ...
}
```

When a class is declared a friend class, all the member functions of the friend class become friend functions.
Since `classB` is a friend class, we can access all members of `classA` from inside `classB`. However, we cannot access members of `ClassB` from inside `classA`. It is because friend relation in C++ is only granted, not taken.

10.2.1 Example 3: C++ friend Class

```
// C++ program to demonstrate the working of friend class

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {
    private:
        int numA;

        // friend class declaration
        friend class ClassB;

    public:
        // constructor to initialize numA to 12
        ClassA() : numA(12) {}
};

class ClassB {
    private:
        int numB;
```

```
    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    // member function to add numA
    // from ClassA and numB from ClassB
    int add() {
        ClassA objectA;
        return objectA.numA + numB;
    }
};

int main() {
    ClassB objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

Output

```
Sum: 13
```

Here, `ClassB` is a friend class of `ClassA` . So, `ClassB` has access to the members of `classA` .

In `ClassB` , we have created a function `add()` that returns the sum of `numA` and `numB` .

Since `ClassB` is a friend class, we can create objects of `ClassA` inside of `ClassB` .

11 Multiple, Multilevel and Hierarchical Inheritance

Inheritance is one of the core feature of an object-oriented programming language. It allows software developers to derive a new class from the existing class. The derived class inherits the features of the base class (existing class). There are various models of inheritance in C++.

11.1 Multilevel Inheritance

In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as `multilevel inheritance` .

```
class A
{
... .. ...
};
class B: public A
{
... .. ...
};
class C: public B
{
... ... ...
};
```

Here, class `B` is derived from the base class `A` and the class `C` is derived from the derived class `B` .

11.1.1 Multilevel Inheritance Example

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

class A
{
    public:
      void display()
      {
          cout<<"Base class content.";
      }
};
```

```
class B : public A
{

};

class C : public B
{

};

int main()
{
    C obj;
    obj.display();
    return 0;
}
```

Output

```
Base class content.
```

In this program, class `C` is derived from class `B` (which is derived from base class `A`).

The `obj` object of class `C` is defined in the `main()` function.

When the `display()` function is called, `display()` in class `A` is executed. It's because there is no `display()` function in class `C` and class `B`.

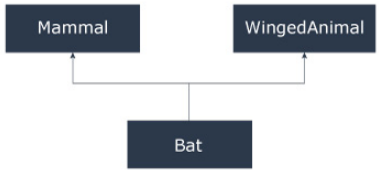
The compiler first looks for the `display()` function in class `C`. Since the function doesn't exist there, it looks for the function in class `B` (as `C` is derived from `B`).

The function also doesn't exist in class `B`, so the compiler looks for it in class `A` (as `B` is derived from `A`).

If `display()` function exists in `C`, the compiler overrides `display()` of class `A` (because of member function overriding).

11.2 Multiple Inheritance

In C++, a class can be derived from more than one parents. For example: A class `Bat` is derived from base classes `Mammal` and `WingedAnimal`. It makes sense because bat is a mammal as well as a winged animal.



11.2.1 Multiple Inheritance Example

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

class Mammal {
  public:
    Mammal()
    {
      cout << "Mammals can give direct birth." << endl;
    }
};

class WingedAnimal {
  public:
    WingedAnimal()
    {
      cout << "Winged animal can flap." << endl;
    }
};

class Bat: public Mammal, public WingedAnimal {

};

int main()
{
    Bat b1;
    return 0;
}
```

Output

```
Mammals can give direct birth.
Winged animal can flap.
```

11.2.2 Ambiguity in Multiple Inheritance

The most obvious problem with multiple inheritance occurs during function overriding.

Suppose, two base classes have a same function which is not overridden in derived class.
If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call.

For example,

```
class base1
{
  public:
     void someFunction( )
     { .... ... .... }
};
class base2
{
    void someFunction( )
     { .... ... .... }
};
class derived : public base1, public base2
{

};

int main()
{
    derived obj;

    obj.someFunction() // Error!
}
```

This problem can be solved using scope resolution function to specify which function to class either `base1 or base2`

```
int main()
{
    obj.base1::someFunction( );  // Function of base1 class is called
    obj.base2::someFunction();   // Function of base2 class is called.
}
```

11.3 Hierarchical Inheritance

If more than one class is inherited from the base class, it's known as `hierarchical inheritance` . In hierarchical inheritance, all features that are common in child classes are included in the base class.

For example: Physics, Chemistry, Biology are derived from Science class.

11.3.1 Syntax of Hierarchical Inheritance

```
class base_class {
     ... .. ...
}

class first_derived_class: public base_class {
     ... .. ...
}

class second_derived_class: public base_class {
     ... .. ...
}

class third_derived_class: public base_class {
     ... .. ...
}
```

12 Polymorphism

Polymorphism is an important concept of object-oriented programming. It simply means more than one form. That is, the same entity (function or operator) behaves differently in different scenarios.

For example, The `+` operator in C++ is used to perform two specific functions. When it is used with numbers (integers and floating-point numbers), it performs addition.

```
int a = 5;
int b = 6;
int sum = a + b;     // sum = 11
```

And when we use the `+` operator with strings, it performs string concatenation. For example,

```
string firstName = "abc ";
string lastName = "xyz";

// name = "abc xyz"
string name = firstName + lastName;
```

We can implement polymorphism in C++ using the following ways:

1. Function Overloading
2. Operator Overloading
3. Function Overriding
4. Virtual Functions

12.1 Function Overloading

In C++, we can use two functions having the same name if they have different parameters (either types or number of arguments).
And, depending upon the number/type of arguments, different functions are called.
For example,

```
// C++ program to overload sum() function

#include <iostream>
using namespace std;

// Function with 2 int parameters
int sum(int num1, int num2) {
    return num1 + num2;
}

// Function with 2 double parameters
double sum(double num1, double num2) {
```

```
    return num1 + num2;
}

// Function with 3 int parameters
int sum(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}

int main() {
    // Call function with 2 int parameters
    cout << "Sum 1 = " << sum(5, 6) << endl;

    // Call function with 2 double parameters
    cout << "Sum 2 = " << sum(5.5, 6.6) << endl;

    // Call function with 3 int parameters
    cout << "Sum 3 = " << sum(5, 6, 7) << endl;

    return 0;
}
```

Output

```
Sum 1 = 11
Sum 2 = 12.1
Sum 3 = 18
```

Here, we have created 3 different `sum()` functions with different parameters (number/type of parameters). And, based on the arguments passed during a function call, a particular `sum()` is called.
It's a **compile-time polymorphism** because the compiler knows which function to execute before the program is compiled.

12.2 Operator Overloading

We can overload an operator as long as we are operating on user-defined types like objects or structures. We cannot use operator overloading for basic types such as `int` , `double` , etc. Operator overloading is basically function overloading, where different operator functions have the same symbol but different operands. And, depending on the operands, different operator functions are executed.
For example,

```
// C++ program to overload ++ when used as prefix

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

```
class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    void operator ++() {
        value = value + 1;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;

    // Call the "void operator ++()" function
    ++count1;

    count1.display();
    return 0;
}
```

Output

```
Count: 6
```

Here, we have overloaded the `++` operator, which operates on objects of `Count` class (object `count1` in this case). We have used this overloaded operator to directly increment the `value` variable of `count1` object by `1` . This is also a **compile-time polymorphism**.

More here: Operator Overloading

12.3 Function Overriding

In C++ inheritance, we can have the same function in the base class as well as its derived classes. When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class. So, different functions are executed depending on the object calling the function. This is known as **function overriding**.
For example,

```cpp
// C++ program to demonstrate function overriding

#include <iostream>
using namespace std;

class Base {
   public:
    virtual void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // Call print() function of Derived class
    derived1.print();

    return 0;
}
```

Output

```
Derived Function
```

Here, we have used a `print()` function in the `Base` class and the same function in the `Derived` class. When we call `print()` using the `Derived` object `derived1`, it overrides the `print()` function of `Base` by executing the `print()` function of the `Derived` class.
It's a **runtime polymorphism** because the function call is not resolved by the compiler, but it is resolved in the runtime instead.

12.4 Virtual Functions

In C++, we may not be able to override functions if we use a pointer of the base class to point to an object of the derived class. Using **virtual functions** in the base class **ensures** that the function can be overridden in these cases. Thus, **virtual functions** actually fall under **function overriding**.
For example,

```cpp
// C++ program to demonstrate the use of virtual functions
```

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

class Base {
   public:
    virtual void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;

    // calls member function of Derived class
    base1->print();

    return 0;
}
```

Output

```
Derived Function
```

Here, we have used a virtual function `print()` in the `Base` class to ensure that it is overridden by the function in the `Derived` class. Virtual functions are **runtime polymorphism**.

12.5 Why Polymorphism?

Polymorphism allows us to create consistent code. For example, Suppose we need to calculate the area of a circle and a square. To do so, we can create a `Shape` class and derive two classes `Circle` and `Square` from it.

In this case, it makes sense to create a function having the same name `calculateArea()` in both the derived classes rather than creating functions with different names, thus making our code more consistent.

13 Virtual Functions

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.
For example, consider the code below:

```
class Base {
   public:
    void print() {
        // code
    }
};

class Derived : public Base {
   public:
    void print() {
        // code
    }
};
```

Later, if we create a pointer of `Base` type to point to an object of `Derived` class and call the `print()` function, it calls the `print()` function of the `Base` class.

In other words, the member function of `Base` is not overridden.

```
int main() {
    Derived derived1;
    Base* base1 = &derived1;

    // calls function of Base class
    base1->print();

    return 0;
}
```

In order to avoid this, we declare the `print()` function of the `Base` class as virtual by using the virtual keyword.

```
class Base {
   public:
    virtual void print() {
        // code
    }
};
```

Virtual functions are an integral part of polymorphism in C++.

13.1 Example 1: C++ virtual Function

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

class Base {
   public:
    virtual void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;

    // calls member function of Derived class
    base1->print();

    return 0;
}
```

Output

```
Derived Function
```

Here, we have declared the `print()` function of `Base` as `virtual`. So, this function is overridden even when we use a pointer of `Base` type that points to the `Derived` object `derived1`.

13.2 C++ Override Keyword

As of C++11 a new identifier `override` is available to use. It's a useful way to avoid run time errors when using virtual functions. Where the keyword: `virtual` is telling the compiler that "I'm available for being overridden" the keyword: `override` is telling the compiler that "I'm overriding a method in my base class".
Example,

```
class Base {
   public:
    virtual void print() {
        // code
    }
};

class Derived : public Base {
   public:
    void print() override {
        // code
    }
};
```

If we use a function prototype in `Derived` class and define that function outside of the class, then we use the following code:

```
class Derived : public Base {
   public:
    // function prototype
    void print() override;
};

// function definition
void Derived::print() {
        // code
    }
```

13.2.1 Using Override

When using virtual functions. it is possible to make mistakes while declaring the member functions of the derived classes. Using the `override` identifier prompts the compiler to display error messages when these mistakes are made. Otherwise, the program will simply compile but the virtual function will not be overridden.

Some of these possible mistakes are:

- **Functions with incorrect names:** For example, if the virtual function in the base class is named `print()` , but we accidentally name the overriding function in the derived class as `pint()` .
- **Functions with different return types:** If the virtual function is, say, of `void` type but the function in the derived class is of `int` type.
- **Functions with different parameters:** If the parameters of the virtual function and the functions in the derived classes don't match.
- No virtual function is declared in the base class.

13.3 Using Virtual Functions

Suppose we have a base class `Animal` and derived classes `Dog` and `Cat` .
Suppose each class has a data member named `type` . Suppose these variables are initialized through their respective constructors.

```
class Animal {
   private:
    string type;
    ... .. ...
    public:
      Animal(): type("Animal") {}
    ... .. ...
};

class Dog : public Animal {
   private:
    string type;
    ... .. ...
    public:
      Animal(): type("Dog") {}
    ... .. ...
};

class Cat : public Animal {
   private:
    string type;
      ... .. ...
    public:
      Animal(): type("Cat") {}
    ... .. ...
};
```

Now, let us suppose that our program requires us to create two `public` functions for each class:

1. `getType()` to return the value of `type`
2. `print()` to print the value of `type`

We could create both these functions in each class separately and override them, which will be long and tedious.
Or we could make `getType()` **virtual** in the `Animal` class, then create a single, separate `print()` function that accepts a pointer of `Animal` type as its argument. We can then use this single function to override the virtual function.

```
class Animal {
    ... .. ...
```

```
   public:
    ... .. ...
    virtual string getType {...}
};

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

void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}
```

This will make the code **shorter**, **cleaner**, and **less repetitive**.

13.4 Using Virtual Functions Again

```
// C++ program to demonstrate the use of virtual function

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

class Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Animal() : type("Animal") {}

    // declare virtual function
    virtual string getType() {
        return type;
    }
};

class Dog : public Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Dog() : type("Dog") {}

    string getType() override {
        return type;
    }
```

```
};

class Cat : public Animal {
   private:
    string type;

   public:
    // constructor to initialize type
    Cat() : type("Cat") {}

    string getType() override {
        return type;
    }
};

void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}

int main() {
    Animal* animal1 = new Animal();
    Animal* dog1 = new Dog();
    Animal* cat1 = new Cat();

    print(animal1);
    print(dog1);
    print(cat1);

    return 0;
}
```

Output

```
Animal: Animal
Animal: Dog
Animal: Cat
```

Here, we have used the virtual function `getType()` and an `Animal` pointer `ani` in order to avoid repeating the `print()` function in every class.

```
void print(Animal* ani) {
    cout << "Animal: " << ani->getType() << endl;
}
```

In `main()`, we have created 3 `Animal` pointers to dynamically create objects of `Animal`, `Dog` and `Cat` classes.

```
// dynamically create objects using Animal pointers
Animal* animal1 = new Animal();
Animal* dog1 = new Dog();
Animal* cat1 = new Cat();
```

We then call the `print()` function using these pointers:

1. When `print(animal1)` is called, the pointer points to an `Animal` object. So, the virtual function in `Animal` class is executed inside of `print()` .
2. When `print(dog1)` is called, the pointer points to a `Dog` object. So, the virtual function is overridden and the function of `Dog` is executed inside of `print()` .
3. When `print(cat1)` is called, the pointer points to a `Cat` object. So, the virtual function is overridden and the function of `Cat` is executed inside of `print()` .

14 Early and Late Binding

Before going to virtual function, let's first have a look at **early binding** and **late binding**.
Binding means matching the function call with the correct function definition by the compiler. It takes place either at compile time or at runtime.

14.1 Early Binding

In early binding, the compiler matches the function call with the correct function definition at compile time. It is also known as **Static Binding** or **Compile-time Binding**. By default, the compiler goes to the function definition which has been called during compile time. So, all the function calls you have studied till now are due to early binding.
You have learned about function overriding in which the base and derived classes have functions with the same name, parameters and return type. In that case also, early binding takes place.
In function overriding, we called the function with the objects of the classes. Now let's try to write the same example but this time calling the functions with the pointer to the base class i.e., refernce to the base class' object.

```
#include <iostream>

using namespace std;

class Animal{
    public:
        void sound(){
            cout << "This is parent class" << endl;
        }
};

class Dog : public Animal{
    public:
        void sound(){
            cout << "Dogs bark" << endl;
        }
};

int main(){

    Animal *a;      // Declare a pointer to an instance of an Animal
    Dog d;          // Instance of a Dog
    a = &d;         // point a to address of d
    a->sound();     // call method sound using pointer a
    return 0;
}
```

Output:

```
This is parent class
```

Why was `Animal` 's method called and not `Dog` 's?

- Obviously `*a` is a pointer to the parent class `Animal` .
- But we pointed `*a` to an instance of the class `Dog` ?!?
- It is allowed since `Dog` is derived from `Animal` , however
-
-

This is due to Early Binding. We know that **a** is a pointer of the parent class referring to the object of the child class. Since early binding takes place at compile-time, therefore when the compiler saw that **a** is a **pointer of the parent class**, it matched the call with the 'sound()' function of the parent class without considering which object the pointer is referring to.

14.2 Late Binding

In the case of late binding, the compiler matches the function call with the correct function definition at runtime. It is also known as **Dynamic Binding** or **Runtime Binding**.
In late binding, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.
By default, early binding takes place. So if by any means we tell the compiler to perform late binding, then the problem in the previous example can be solved.
This can be achieved by declaring a **virtual function**.

14.3 Virtual function

Virtual Function is a member function of the base class which is overridden in the derived class. The compiler performs **late binding** on this function.
To make a function virtual, we write the keyword **virtual** before the function definition.

```
#include <iostream>

using namespace std;

class Animals
{
    public:
        virtual void sound()
        {
            cout << "This is parent class" << endl;
        }
};
```

```
class Dogs : public Animals
{
    public:
        void sound()
        {
            cout << "Dogs bark" << endl;
        }
};

int main()
{
    Animals *a;
    Dogs d;
    a= &d;
    a -> sound();
    return 0;
}
```

Output:

```
Dogs bark
```

Since the function **sound()** of the base class is made virtual, the compiler now performs late binding for this function. Now, the function call will be matched to the function definition at runtime. Since the compiler now identifies pointer **a** as referring to the object 'd' of the derived class Dogs, it will call the sound() function of the class Dogs.

If we declare a member function in a base class as a virtual function, then that function automatically becomes virtual in all of its derived classes.
If we make any function inside a base class virtual, then that function becomes virtual in all its derived classes. This means that we don't need to declare that function as virtual separately in its derived classes.

We can also call private function of derived class from a base class pointer by declaring that function in the base class as virtual.
Compiler checks if the members of a class are private, public or protected only at compile time and not at runtime. Since our function is being called at runtime, so we can call any type of function, private or public as shown in the following example.

```
#include <iostream>

using namespace std;

class Animals
{
    public:
        virtual void sound()
        {
```

```
            cout << "This is parent class" << endl;
        }
};

class Dogs : public Animals
{
    private:
        virtual void sound()
        {
            cout << "Dogs bark" << endl;
        }
};

int main()
{
    Animals *a;
    Dogs b;
    a = &b;
    a->sound();
    return 0;
}
```

Output
Dogs bark
Since the same function (virtual function) having different definitions in different classes is called depending on the type of object that calls the function, this is also a part of **Polymorphism**.

14.4 Pure Virtual Function

Pure virtual function is a virtual function which has no definition. Pure virtual functions are also called **abstract functions**.
To create a pure virtual function, we assign a value **0** to the function as follows.
virtual void sound() = 0;
Here sound() is a pure virtual area.

14.5 Abstract Class

An **abstract class** is a class whose instances (objects) can't be made. We can only make objects of its subclass (if they are not abstract). Abstract class is also known as **abstract base class**.
An abstract class has at least one abstract function (pure virtual function).
Let's look at an example of abstract class.
Suppose there are some employees working in a firm. The firm hires only two types of employees- either driver or developer. Now, you have to develop a software to store information about them.

So, here is an idea - There is no need to make objects of employee class. We will make objects to only driver or developer. Also, both must have some salary. So, there must be a common function to know about salary.
This need will be best accomplished with **abstract class**.
So, we can make 'Employee' an abstract class and 'Developer' and 'Driver' its subclasses.

```
#include <iostream>

using namespace std;

class Employee                      //  abstract base class
{
    virtual int getSalary() = 0;    // pure virtual function
};

class Developer : public Employee
{
    int salary;
    public:
        Developer(int s)
        {
            salary = s;
        }
        int getSalary()
        {
            return salary;
        }
};

class Driver : public Employee
{
    int salary;
    public:
        Driver(int t)
        {
            salary = t;
        }
        int getSalary()
        {
            return salary;
        }
};

int main()
{
    Developer d1(5000);
    Driver d2(3000);
    int a, b;
```

```
    a = d1.getSalary();
    b = d2.getSalary();
    cout << "Salary of Developer : " << a << endl;
    cout << "Salary of Driver : " << b << endl;
    return 0;
}
```

Output
Salary of Developer : 5000
Salary of Driver : 3000
The **getSalary()** function in the class **Employee** is a pure virtual function. Since the Employee class contains this pure virtual function, therefore it is an **abstract base class**.
Since the abstract function is defined in the subclasses, therefore the function 'getSalary()' is defined in both the subclasses of the class Employee.
You must have understood the rest of the code.

Subclasses of an abstract base class must define the abstract method, otherwise, they will also become abstract classes.

In an abstract class, we can also have other functions and variables apart from pure virtual function.
Let's see one more example of abstract base class.

```
#include <iostream>

using namespace std;

class Animals
{
    public:
        virtual void sound() = 0;
};

class Dogs
{
    public:
        void sound()
        {
            cout << "Dogs bark" << endl;
        }
};

class Cats
{
    public:
        void sound()
        {
            cout << "Cats meow" << endl;
```

```
        }
};

class Pigs
{
    public:
        void sound()
        {
            cout << "Pigs snort" << endl;
        }
};

int main()
{
    Dogs d;
    Cats c;
    Pigs p;
    d.sound();
    c.sound();
    p.sound();
    return 0;
}
```

Output
Dogs bark
Cats meow
Pigs snort

14.6 Interface

Interface or **Interface class** is a class which is the same as abstract class with a difference that all its functions are pure virtual and it has no member variables. Its derived classes must implement each of its virtual functions i.e., provide definition to each of the pure virtual functions of the base class.
Like an abstract class, we can't create objects of an interface.
We can also say that interface is an abstract class with no member variables and all its member functions pure virtual.

Name of an interface class often begins with the letter **I**.
Let's see an example of it.

```
class IShape
{
   public:
      virtual getArea() = 0;
      virtual getPerimeter() = 0;
};
```

IShape is an interface because it contains only pure virtual functions.

```
#include <iostream>

using namespace std;

class IShape
{
    public:
        virtual int getArea() = 0;
        virtual int getPerimeter() = 0;
};

class Rectangle : public IShape
{
    int length;
    int breadth;
    public:
        Rectangle(int l, int b)
        {
            length = l;
            breadth = b;
        }
        int getArea()
        {
            return length * breadth;
        }
        int getPerimeter()
        {
            return 2*(length + breadth);
        }
};

class Square : public IShape
{
    int side;
    public:
        Square(int a)
        {
            side = a;
        }
        int getArea()
        {
            return side * side;
        }
        int getPerimeter()
        {
            return 4 * side;
        }
};
```

```
int main()
{
    Rectangle rt(7, 4);
    Square s(4);
    cout << "Rectangle :" << endl;
    cout << "Area : " << rt.getArea() << " Perimeter : " << rt.getPerimeter() << endl;
    cout << "Square :" << endl;
    cout << "Area : " << s.getArea() << " Perimeter : " << s.getPerimeter() << endl;
    return 0;
}
```

Output
Rectangle :
Area : 28 Perimeter : 22
Square :
Area : 16 Perimeter : 16
So we just saw that IShape is an interface with two pure virtual functions. These virtual functions are implemented (defined) in its subclasses Rectangle and Square according to their requirements.
So, an interface is just an abstract class with all pure virtual methods.

15 Internal and External Linkage

Original Author: Peter Goldsborough

A *translation unit* refers to an implementation (`.c/.cpp`) file and all header (`.h/.hpp`) files it includes. If an object or function inside such a translation unit has *internal linkage*, then that specific symbol is only visible to the linker within that translation unit. If an object or function has *external linkage*, the linker can also see it when processing other translation units. The `static` keyword, when used in the global namespace, forces a symbol to have internal linkage. The `extern` keyword results in a symbol having external linkage.

The compiler defaults the linkage of symbols such that:

- `Non-const` global variables have `external` linkage by default
- `Const` global variables have `internal` linkage by default
- Functions have `external` linkage by default

15.1 Basics

Lets first cover two rudimentary concepts that we'll need to properly discuss linkage:

1. The difference between a declaration and a definition
2. Translation Units

Also, just a quick word on naming: we'll use the term *symbol* to refer to any kind of "code entity" that a linker works with, i.e. *variables* and *functions* (and also classes/structs, but we won't talk much about those).

15.1.1 Declaration vs. Definition

Lets quickly discuss the difference between `declaring` a symbol and `defining` a symbol: A `declaration` tells the compiler about the existence of a certain symbol and makes it possible to refer to that symbol everywhere where the explicit memory address or required storage of that symbol is not required. A `definition` tells the compiler what the body of a function contains or how much memory it must allocate for a variable.

Situations where a declaration is not sufficient to the compiler are, for example, when a data member of a class is of reference or value (as in, neither reference nor pointer) type. At the same time, it is always allowed to have pointers to a declared (but not defined) type, because pointers require fixed memory capacity (e.g. 8 bytes on 64-bit systems) and do not depend on the type pointed to. When you dereference that pointer, the definition does become necessary. Also, for function `declarations` , all parameters (no matter whether taken by value, reference or pointer) and the return type need only be declared and not defined. Definitions of parameter and return value types only become necessary for the function definition.

Functions

The difference between the declaration and definition of a function is fairly obvious:

```
int f();               // declaration
int f() { return 42; } // definition
```

Variables

For variables, it is a bit different. Declaration and definition are usually not explicitly separate. *Most importantly*, this:

```
int x;
```

Does not just declare `x`, but also define it. In this case, by calling the default constructor of `int`. (As an aside, in C++, as opposed to Java, the constructor of primitive types (such as `int`) does not default-initialize (in C++ lingo: *value initialize*) the variable to `0`. The value of `x` above will be whatever garbage lay at the memory address allocated for it by the compiler.)
You can, however, explicitly separate the declaration of a variable from its definition by using the `extern` keyword:

```
extern int x; // declaration
int x = 42;   // definition
```

However, when `extern` is prepended to the declaration and an initialization is provided as well, then the expression turns into a definition and the `extern` keyword essentially becomes useless:

```
extern int x = 5; // is the same thing as
int x = 5;
```

Forward Declaring

In C++ there exists the concept of *forward declaring* a symbol. What we mean by this is that we declare the type and name of a symbol so that we can use it where its definition is not required. By doing so, we don't have to include the full definition of a symbol (usually a header file) when it is not explicitly necessary. This way, we reduce dependency on the file containing the definition. The main advantage of this is that when the file containing the definition changes, the file where we forward declared that symbol does not need to be re-compiled (and therefore, also not all further files including it).

Example

Say we have a function declaration (also called *prototype*) for `f`, taking an object of type `Class` by value:

```
// file.hpp

void f(Class object);
```

Now, the naïve thing to do would be to include `Class` 's definition right away. But because we only declare `f` here, it is sufficient to provide the compiler with a declaration of `Class` . This way, the compiler can identify the function by its prototype, but we can remove the dependency of `file.hpp` on the file containing the definition of `Class` , say `class.hpp` :

```
// file.hpp

class Class;

void f(Class object);
```

Now, say, we include `file.hpp` in 100 other files. And say we change `Class` 's definition in `class.hpp` . If we had included `class.hpp` in `file.hpp` , `file.hpp` and all 100 files including it would have to be recompiled. By forward declaring `Class` , the only files requiring recompilation are `class.hpp` and `file.cpp` (assuming that's where `f` is defined).

Usage Frequency

One very important difference between declarations and definitions is that a symbol may be *declared many times, but defined only once*. For example, you can forward declare a function or class however often you want, but you may only ever have one definition for it. This is called the one definition rule. Therefore, this is valid C++:

```
int f();
int f();
int f();
int f();
int f();
int f();
int f() { return 5; }
```

While this isn't:

```
int f() { return 6; }
int f() { return 9; }
```

15.1.2 Translation Units

Programmers usually deal with header files and implementation files. Compilers don't – they deal with *translation units* (TUs), sometimes referred to as *compilation units*. The definition of such a translation unit is very simple: Any file, fed to the compiler, *after it has been pre-processed*. In detail, this means that it is the file resulting from the pre-processor expanding macros, conditionally including source code depending on `#ifdef` and `#ifndef` statements and copy-pasting any `#include` ed files.

Given these files:
`header.hpp` :

```
#ifndef HEADER_HPP
#define HEADER_HPP

#define VALUE 5

#ifndef VALUE
struct Foo { private: int ryan; };
#endif

int strlen(const char* string);

#endif /* HEADER_HPP */
```

`program.cpp` :

```
#include "header.hpp"

int strlen(const char* string)
{
    int length = 0;

    while(string[length]) ++length;

    return length + VALUE;
}
```

The pre-processor will produce the following translation unit, which is then fed to the compiler:

```
int strlen(const char* string);

int strlen(const char* string)
{
    int length = 0;

    while(string[length]) ++length;

    return length + 5;
}
```

15.2 Linkage

Now that we've covered the basics, we can deal with linkage. In general, linkage will refer to the visibility of symbols to the linker when processing files. Linkage can be either *internal* or *external*.

15.2.1 External Linkage

When a symbol (variable or function) has external linkage, that means that that symbol is visible to the linker from other files, i.e. it is "globally" visible and can be shared between translation units.

In practice, this means that you must define such a symbol in a place where it will end up in one and only one translation unit, typically an implementation file (`.c` / `.cpp`), such that it has only one visible definition. If you were to define such a symbol on the spot, along with declaring it, or to place its definition in the same file you declare it, you run the risk of making your linker very angry. As soon as you include that file in more than one implementation file, such that its definition ends up in more than one translation unit, your linker will start crying.
In C and C++, the `extern` keyword (explicitly) declares a symbol to have external linkage:

```
extern int x;
extern void f(const std::string& argument);
```

Both of these symbols have external linkage. Above it was mentioned that `const` global variables have *internal* linkage by default, and non- `const` global variables have *external* linkage by default. That means that `int x;` is the same as `extern int x;` , right? Not quite. `int x;` is actually the same as `extern int x{};` (using C++11 uniform/brace initialization syntax to avoid the most vexing parse), as `int x;` not only declares, but also defines `x` . Therefore, not prepending `extern` to `int x;` in the global scope is just as bad as also defining a variable when declaring it as `extern` :

```
int x;          // is the same as
extern int x{}; // which will both likely cause linker errors.

extern int x;   // while this only declares the integer, which is ok.
```

Example Badness

Let's declare a function `f` with external linkage in `file.hpp` and also define it in the same file:

```
// file.hpp

#ifndef FILE_HPP
#define FILE_HPP

extern int f(int x);

/* ... */

int f(int) { return x + 1; }

/* ... */

#endif /* FILE_HPP */
```

Note that prepending `extern` here is redundant, as all functions are implicitly `extern` , and separating the declaration from the definition here is also unnecessary. So let's just quickly rewrite this as:

```cpp
// file.hpp

#ifndef FILE_HPP
#define FILE_HPP

int f(int) { return x + 1; }

#endif /* FILE_HPP */
```

This is code one would be inclined to write before reading this article or after reading it but under influence of alcohol or strong drugs (e.g. pop tarts).

So let's see why this is bad. We'll now have two implementation files: `a.cpp` and `b.cpp`, both including this `file.hpp`:

```cpp
// a.cpp

#include "file.hpp"

/* ... */

// b.cpp

#include "file.hpp"

/* ... */
```

Now let the compiler do its job and generate two translation units for the two implementation files above (remember that `#include`ing means to literally copy-paste):

```cpp
// TU A, from a.cpp

int f(int) { return x + 1; }

/* ... */

// TU B, from b.cpp

int f(int) { return x + 1; }

/* ... */
```

At this point, the linker will step in (linking comes after compilation). The linker will pick up the symbol `f` and look for definitions. Because it's the linker's lucky day, it will even find two! One in TU A and one in TU B. The linker will be so happy, it'll stop and tell you in a way similar to this:

```
duplicate symbol __Z1fv in:
/path/to/a.o
/path/to/b.o
```

The linker found two definitions for the same symbol `f` . Because it had external linkage, `f` was visible to the linker when processing both TU A and TU B. Naturally, this violates the One-Definition-Rule, so this causes a linker error. More specifically, this is when you get a *duplicate symbol* error, which is the one you'll get most often along with an *undefined symbol* error (if we had only ever declared, but never defined `f`).

Usage

A common use case for declaring variables explicitly `extern` are global variables. For example, say you are working on a self-baking cake. There may be certain global system variables connected with self-baking cakes that you need to access in various places throughout your program. Let's say the clock-rate of the edible chip inside your cake. Such a value would naturally be required in many, many places to make all the chocolate electronics work synchronously. The C (evil) way of declaring such a global variable would be a macro:

```
#define CLK 1000000
```

A C++ programmer, naturally despising macros, would rather use real code. So you could do this:

```
// global.hpp

namespace Global
{
    extern unsigned int clock_rate;
}

// global.cpp
namespace Global
{
    unsigned int clock_rate = 1000000;
}
```

(As a modern C++ programmer, you might also want to take advantage of (separator literals)[http://www.informit.co
`unsigned int clock_rate = 1'000'000;`)

15.2.2 Internal Linkage

When a symbol has *internal linkage*, it will only be visible within the current translation unit. Do not confuse the term *visible* here with access rights like `private` . *Visibility* here means that the linker will only be able to use this symbol when processing the translation unit in which the symbol was declared, and not later (as with symbols with external linkage). In practice, this means that when you declare a symbol to have internal linkage in a header file, each translation unit you include this file in will get *its own unique copy of that symbol*. I.e. it will be as if you redefined each such symbol in every translation unit. For objects, this means that the compiler will literally allocate an entirely new, unique copy for each translation unit, which can obviously incur high memory costs.

To declare a symbol with internal linkage, C and C++ provide the `static` keyword. Its usage here is entirely separate from its usage in classes or functions (or, generally, any block).

Example Here an example:
`header.hpp` :

```
static int variable = 42;
```

`file1.hpp` :

```
void function1();
```

`file2.hpp` :

```
void function2();
```

`file1.cpp` :

```
#include "header.hpp"

void function1() { variable = 10; }
```

`file2.cpp` :

```
#include "header.hpp"

void function2() { variable = 123; }
```

`main.cpp` :

```
#include "header.hpp"
#include "file1.hpp"
#include "file2.hpp"

#include <iostream>

auto main() -> int
{
    function1();
    function2();

    std::cout << variable << std::endl;
}
```

Because `variable` has internal linkage, each translation unit that includes `header.hpp` gets its own unique copy of `variable` . Here, there are three translation units:

1. file1.cpp
2. file2.cpp
3. main.cpp

When `function1` is called, `file1.cpp` 's copy of `variable` is set to 10. When `function2` is called, `file2.cpp` 's copy of `variable` is set to 123. However, the value printed out in `main.cpp` is `variable` , unchanged: 42.

Anonymous Namespaces

In C++, there exists another way to declare one or more symbols to have internal linkage: anonymous namespaces. Such a namespace ensures that the symbols declared inside it are visible only within the current translation unit. It is, in essence, just a way to declare many symbols as `static` . In fact, for a while, the `static` keyword for the use of declaring a symbol to have internal linkage was deprecated in favor of anonymous namespaces. However, it was recently *undeprecated*, because it is useful to declare a single variable or function to have internal linkage. There are also a few minor differences which I won't go into here.
In any case, this:

```
namespace { int variable = 0; }
```

does (almost) the same thing as this:

```
static int variable = 0;
```

Usage

So when and why would one make use of internal linkage? For objects, it is probably most often a very bad idea to make use of it. The memory cost can be very high for large objects given that each translation unit gets its own copy. But mainly, it can really just cause odd, unexpected behavior. Imagine you had a singleton (a class of which you instantiate only a single instance), and would suddenly end up having multiple instances of your "singleton" (one for every translation unit).
However, one interesting use case could be to hide translation-unit-local helper functions from the global scope. Imagine you have a helper function `foo` in your `file1.hpp` which you use in `file1.cpp` , but then you also have a helper function `foo` in your `file2.hpp` which you use in `file2.cpp` . The first `foo` does something completely different than the second `foo` , but you cannot think of a better name for them. So, you can declare them both `static` . Unless you include both `file1.hpp` and `file2.hpp` in some same translation unit, this will hide the respective `foo` s from each other. If you don't declare them `static` , they will implicitly have external linkage and the first `foo` 's definition will collide with the second `foo` s definition and cause a linker error due to a violation of the one-definition-rule.

15.3 References

- http://stackoverflow.com/questions/154469/unnamed-anonymous-namespaces-vs-static-functions
- http://stackoverflow.com/questions/4726570/deprecation-of-the-static-keyword-no-more
- http://www.geeksforgeeks.org/understanding-extern-keyword-in-c/