Classes and Objects

Topics

- Abstract Data Types
- Object-Oriented Programming
- Introduction to Classes
- Introduction to Objects
- Defining Member Functions
- Constructors
- Destructors
- Private Member Functions
- Object Relations
  - Inheritance

1. ADT and OOP

Abstract Data Types (ADT)

- Programmer-created data types that specify
  - legal values that can be stored
  - operations that can be done on the values

- The user of an abstract data type (ADT) does not need to know any implementation details
  - (*e.g.*, how the data is stored or how the operations on it are carried out)

Abstraction and Data Types

- Abstraction: definition that captures general characteristics without details
  - An abstract triangle is a 3-sided polygon.
  - A specific triangle may be scalene, isosceles, or equilateral
- Data Type: defines kind of values that can be stored & operations that can be performed on it

Object-Oriented Programming (OOP)

- Procedural programming uses variables to store data, focuses on the processes/ functions that occur in a program. Data and functions are separate and distinct.
- Object-oriented programming is based on objects that encapsulate the data and the functions that operate on it.

Procedural V.S. Object-oriented

- Which one is more natural?

  driveTo(you, work);

  you.driveTo(work);

- This not only reads more clearly, it also makes it clearer who the subject is (you) and what behavior is being invoked (driving somewhere). Rather than being focused on writing functions, we're focused on defining objects that have a well-defined set of behaviors. This is why the paradigm is called "object-oriented".
- This allows programs to be written in a more modular fashion, which makes them easier to write and understand, and also provides a higher degree of code-reusability. These objects also provide a more intuitive way to work with our data by allowing us to define how we interact with the objects, and how they interact with other objects.

Structure v.s. Class

```cpp
#include <iostream>

struct DateStruct
{
    int year;
    int month;
    int day;
};

void print(DateStruct &date)
{
    std::cout << date.year << "/" <<
        date.month << "/" << date.day;
}

int main()
{
    // use uniform initialization
    DateStruct today { 2020, 10, 14 };

    // use member selection operator to
    //select a member of the struct
    today.day = 16;
    print(today);

    return 0;
}
```

```cpp
#include <iostream>

class DateClass
{
public:
    int m_year;
    int m_month;
    int m_day;

    void print()
    {
        std::cout << m_year <<
         "/" << m_month << "/" << m_day;
    }
};

int main()
{
    DateClass today { 2020, 10, 14 };
    // use member selection operator to
    //select a member variable of the class
    today.m_day = 16;
    // use member selection operator to call
    // a member function of the class
    today.print();

    return 0;
}
```

Structure v.s. Class

- However, there are a few differences. In the DateStruct version of print() from the example above, we needed to pass the struct itself to the print() function as the first parameter. Otherwise, print() wouldn't know what DateStruct we wanted to use. We then had to reference this parameter inside the function explicitly.

- Member functions work slightly differently: All member function calls must be associated with an object of the class. When we call "today.print()", we're telling the compiler to call the print() member function, associated with the today object.

Example

```
#include <iostream>
#include <string>

class Employee
{
public:
    std::string m_name;
    int m_id;
    double m_wage;

    // Print employee information to the screen
    void print()
    {
        std::cout << "Name: " << m_name <<
                "  Id: " << m_id <<
                "  Wage: $" << m_wage << '\n';
    }
};

int main()
{
    // Declare two employees
    Employee alex { "Alex", 1, 25.00 };
    Employee joe { "Joe", 2, 22.25 };

    // Print out the employee information
    alex.print();
    joe.print();

    return 0;
}
```

This produces the output:

```
Name: Alex  Id: 1  Wage: $25
Name: Joe  Id: 2  Wage: $22.25
```

You have already been using classes without knowing it

- C++ standard library is full of classes that have been created for your benefit. std::string, std::vector, and std::array are all class types.

```cpp
#include <string>
#include <array>
#include <vector>
#include <iostream>

int main()
{
    std::string s { "Hello, world!" }; // instantiate a string class object
    std::array<int, 3> a { 1, 2, 3 }; // instantiate an array class object
    std::vector<double> v { 1.1, 2.2, 3.3 }; // instantiate a vector class object

    std::cout << "length: " << s.length() << '\n'; // call a member function

    return 0;
}
```

OOP Terminology

- **object**: software entity that combines data & functions that act on the data in a single unit
- **attributes**: the data items of an object, stored in **member variables**
- **member functions (methods)**: procedures/ functions that act on attributes of the class

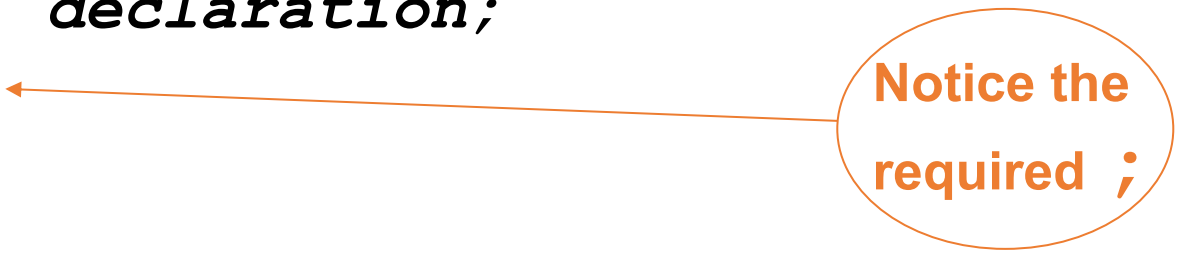
More Object-Oriented Programming Terminology

- **data hiding**: restricting access to certain members of an object. Intent is to allow only member functions to directly access modify object's data
- **encapsulation**: the bundling of an object's data & procedures into a single entity

Introduction to Classes

- **Class**: a programmer-defined data type used to define objects
- It is a template for creating objects
- Class declaration format:

```
class className
{
   declaration;
   declaration;
};
```



2. Access Specifiers (public/private)

Public vs Private access specifiers

- Statements used to control access to members of the class.
- Each member is declared to be either

`public`: can be accessed by functions outside of the class

Or

`private`: can only be called by or accessed by functions that are members of the class (default setting)

Attributes are private by default

```
// members are public by default
struct DateStruct
{
    int month;
    int day;
    int year;
};

int main()
{
    DateStruct date;
    date.month = 10;
    date.day = 14;
    date.year= 2020;

    return 0;
}
```

```
// members are private by default
class DateClass
{
    int m_month;
    int m_day;
    int m_year;
};

int main()
{
    DateClass date;
    date.m_month = 10; // error
    date.m_day = 14; // error
    date.m_year = 2020; // error

    return 0;
}
```

- By default, all members of a class are private. **Private members** are members of a class that can only be accessed by other members of the class. Because main() is not a member of DateClass, it does not have access to date's private members.

Public access specifier

- Although class members are private by default, we can make them public by using the public keyword

```
class DateClass
{
public: // note use of public keyword here, and the colon
    int m_month; // public, can be accessed by anyone
    int m_day; // public, can be accessed by anyone
    int m_year; // public, can be accessed by anyone
};

int main()
{
    DateClass date;
    date.m_month = 10; // okay because m_month is public
    date.m_day = 14;  // okay because m_day is public
    date.m_year = 2020;  // okay because m_year is public

    return 0;
}
```

- Because DateClass's members are now public, they can be accessed directly by main()

Mixing access specifiers

- *Rule: Make member variables private, and member functions public, unless you have a good reason not to*.
- Let's take a look at an example of a class that uses both private and public access.

```
#include <iostream>

class DateClass // members are private by default
{
    int m_month; // private by default, can only be accessed by other members
    int m_day; // private by default, can only be accessed by other members
    int m_year; // private by default, can only be accessed by other members

public:
    void setDate(int month, int day, int year) // public, can be accessed by anyone
    {
        // setDate() can access the private members of the class because it is a member of the class it
self
        m_month = month;
        m_day = day;
        m_year = year;
    }

    void print() // public, can be accessed by anyone
    {
        std::cout << m_month << "/" << m_day << "/" << m_year;
    }
};

int main()
{
    DateClass date;
    date.setDate(10, 14, 2020); // okay, because setDate() is public
    date.print(); // okay, because print() is public

    return 0;
}
```

This program prints:

```
10/14/2020
```

Defining Member Functions

- Member functions are part of class declaration
- Can place entire function definition inside the class declaration (file)

  or

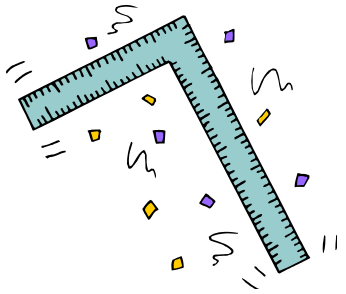
- Can place just the prototype inside the class declaration & write the function definition after the class (in a separate file)

Defining Member Functions Inside the Class Declaration

- Member functions defined inside the class declaration are called inline functions
- Only very short functions, like the one below, should be inline functions

```
int getSide()
{ return side; }
```

Class Example



```
class Square
{
   private:
      int side;

   public:
      void setSide(int s)
      { side = s; }

      int getSide()
      { return side; }
};
```

Access specifiers

Defining Member Functions After the Class Declaration

- Put a function prototype in the class declaration
- In the function definition, precede function name with class name and scope resolution operator (`::`)

```
int Square::getSide()
{
   return side;
}
```

Example: Defining Member Functions After the Class Declaration

```
// C++ program to show that scope resolution operator :: is used
// to define a function outside a class
#include<iostream>
using namespace std;

class A
{
public:

   // Only declaration
   void fun();
};

// Definition outside class using ::
void A::fun()
{
   cout << "fun() called";
}

int main()
{
   A a;
   a.fun();
   return 0;
}
```

3. Access functions and encapsulation

Why make member variables private?

- They provide a simple interface for you to use (a button, a steering wheel, etc…) to perform an action. However, how these devices actually operate is hidden away from you.
- This separation of interface and implementation is extremely useful because it allows us to use objects without understanding how they work.
- This vastly reduces the complexity of using these objects, and increases the number of objects we're capable of interacting with.
- For similar reasons, the separation of implementation and interface is useful in programming.

Encapsulation

- In object-oriented programming, **Encapsulation** (also called **information hiding**) is the process of keeping the details about how an object is implemented hidden away from users of the object.
- Instead, users of the object access the object through a public interface.
- In this way, users are able to use the object without having to understand how it is implemented.
- In C++, encapsulation is implemented via access specifiers. **Typically**, all **member variables** of the class are made private (hiding the implementation details), and most **member functions** are made public (exposing an interface for the user).

Encapsulation Benefit 1

- **Encapsulated classes are easier to use and reduce the complexity of your programs.**

- With a fully encapsulated class, you only need to know what member functions are publicly available to use the class, what arguments they take, and what values they return. It doesn't matter how the class was implemented internally.
- This dramatically reduces the complexity of your programs, and also reduces mistakes.

Encapsulation Benefit 2

- **Encapsulated classes help protect your data and prevent misuse.**
- Global variables are dangerous because you don't have strict control over who has access to the global variable, or how they use it.

```
class MyString
{
    char *m_string; // we'll dynamically allocate our string here
    int m_length; // we need to keep track of the string length
};
```

- m_length should always equal the length of the string held by m_string.

If m_length were public, anybody could change the length of the string without changing m_string (or vice-versa). This would put the class into an inconsistent state.

- By making both m_length and m_string private, users are forced to use whatever public member functions are available to work with the class (and those member functions can ensure that m_length and m_string are always set appropriately).

Encapsulation Benefit 3

- **Encapsulated classes can lower the influence of change.**
- Consider this simple example:

```
#include <iostream>

class Something
{
public:
    int m_value1;
    int m_value2;
    int m_value3;
};

int main()
{
    Something something;
    something.m_value1 = 5;
    std::cout << something.m_value1 << '\n';
}
```

While this program works fine, what would happen if we decided to rename m_value1, or change its type? We'd break not only this program, but likely most of the programs that use class Something as well!

Cont.

- Encapsulation gives us the ability to change how classes are implemented without breaking all of the programs that use them as well.
- Here is the encapsulated version of this class that uses functions to access m_value1:

```
#include <iostream>

class Something
{
private:
    int m_value1;
    int m_value2;
    int m_value3;

public:
    void setValue1(int value) { m_value1 = value; }
    int getValue1() { return m_value1; }
};

int main()
{
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
}
```

Now, let's change the class's implementation:

Cont.

```
#include <iostream>

class Something
{
private:
    int m_value[3]; // note: we changed the implementation of this class!

public:
    // We have to update any member functions to reflect the new implementation
    void setValue1(int value) { m_value[0] = value; }
    int getValue1() { return m_value[0]; }
};

int main()
{
    // But our program still works just fine!
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
}
```

Note that because we did not alter the prototypes of any functions in our class's public interface, our program that uses the class continues to work without any changes.

Encapsulation Benefit 4

- **Encapsulated classes are easier to debug.**

- Encapsulation helps you debug the program when something goes wrong. Often when a program does not work correctly, it is because one of our member variables has an incorrect value. If everyone is able to access the variable directly, tracking down which piece of code modified the variable can be difficult (it could be any of them, and you'll need to breakpoint them all to figure out which). However, if everybody has to call the same public function to modify a value, then you can simply breakpoint that function and watch as each caller changes the value until you see where it goes wrong.

4. Constructor

Uniform Initialization

- When all members of a class (or struct) are public, we can initialize the class (or struct) directly using an initialization list or uniform initialization:

```
class Foo
{
public:
    int m_x;
    int m_y;
};

int main()
{
    Foo foo1 = { 4, 5 }; // initialization list
    Foo foo2 { 6, 7 }; // uniform initialization (C++11)

    return 0;
}
```

- However, as soon as we make any member variables private, we're no longer able to initialize classes in this way. It does make sense: if you can't directly access a variable (because it's private), you shouldn't be able to directly initialize it.

Constructors

- A **constructor** is a member function used to initialize data members of a class
- Called automatically when an object is created
  - That is, declared
  - Cannot call explicitly
- Must be a **`public`** member function
- Must be named the same as the class
- Must have no return type

Default Constructors

- A constructor that takes no parameters (or has parameters that all have default values) is called a **default constructor**. The default constructor is called if no user-provided initialization values are provided.

```
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction() // default constructor
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};

int main()
{
    Fraction frac; // Since no arguments, calls Fraction() default constructor
    std::cout << frac.getNumerator() << "/" << frac.getDenominator() << '\n';

    return 0;
}
```

Default Constructors

```
#include <cassert>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction() // default constructor
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    // Constructor with two parameters, one parameter having a default value
    Fraction(int numerator, int denominator=1)
    {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};
```

- Note that we now have two constructors: a default constructor that will be called in the default case, and a second constructor that takes two parameters.

These two constructors can coexist peacefully in the same class due to function overloading.

Reducing your constructors

```
#include <cassert>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1)
    {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
};
```

An implicitly generated default constructor

- Consider the following class:

```
class Date
{
private:
    int m_year = 1900;
    int m_month = 1;
    int m_day = 1;
};
```

- This class has no constructor. Therefore, the compiler will generate a constructor that behaves identically to the following:

```
class Date
{
private:
    int m_year = 1900;
    int m_month = 1;
    int m_day = 1;

public:
    Date() // implicitly generated constructor
    {
    }
};
```

An implicitly generated default constructor

- Although you can't see the implicitly generated constructor, you can prove it exists:

```
class Date
{
private:
    int m_year = 1900;
    int m_month = 1;
    int m_day = 1;

    // No constructor provided, so C++ creates a public default constructor for us
};

int main()
{
    Date date; // calls implicit constructor

    return 0;
}
```

- The above code compiles, because date object will use the implicit constructor (which is public).

Common Rules

- If your class has any other constructors, the implicitly generated constructor will not be provided.

```
class Date
{
private:
    int m_year = 1900;
    int m_month = 1;
    int m_day = 1;

public:
    Date(int year, int month, int day) // normal non-default constructor
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    // No implicit constructor provided because we already defined our own constructor
};

int main()
{
    Date date; // error: Can't instantiate object because default constructor doesn't exist and the compiler won't generate one
    Date today(2020, 10, 14); // today is initialized to Oct 14th, 2020

    return 0;
}
```

- Generally speaking, it's a good idea to always provide at least one constructor in your class. This explicitly allows you to control how objects of your class are allowed to be created, and will prevent your class from potentially breaking later when you add other constructors.

Classes containing classes

- A class may contain other classes as member variables. By default, when the outer class is constructed, the member variables will have their default constructors called. This happens before the body of the constructor executes.

```
#include <iostream>

class A
{
public:
    A() { std::cout << "A\n"; }
};

class B
{
private:
    A m_a; // B contains A as a member variable

public:
    B() { std::cout << "B\n"; }
};

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

This prints:

```
A
B
```

- This makes sense when you think about it, as the B() constructor may want to use variable m_a -- so m_a had better be initialized first!

Constructor member initializer lists

- Initialize variables in three ways: copy, direct, and via uniform initialization (C++11 only).

```
int value1 = 1; // copy initialization
double value2(2.2); // direct initialization
char value3 {'c'}; // uniform initialization
```

Rule: Favor direct initialization over copy initialization

Rule: Favor uniform initialization over direct initialization if your compiler is C++11 compatible

```
class Something
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Something()
    {
        // These are all assignments, not initializations
        m_value1 = 1;
        m_value2 = 2.2;
        m_value3 = 'c';
    }
};
```

Assignment, not initialization

```
class Something
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Something() : m_value1(1), m_value2(2.2), m_value3('c') // directly initialize our member variables
    {
    // No need for assignment here
    }

    void print()
    {
        std::cout << "Something(" << m_value1 << ", " << m_value2 << ", " << m_value3 << ")\n";
    }
};

int main()
{
    Something something;
    something.print();
    return 0;
}
```

directly initialize member variables

Rule: Use member initializer lists to initialize your class member variables instead of assignment.

Constructor member initializer lists

- If the initializer list doesn't fit on the same line as the function name, then it should go indented on the next line.

```
class Something
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Something(int value1, double value2, char value3='c') // this line already has a lot of stuff on it
        : m_value1(value1), m_value2(value2), m_value3(value3) // so we can put everything indented on next line
    {
    }

};
```

Non-static member initialization

- Starting with C++11, it's possible to give normal class member variables (those that don't use the static keyword) a default initialization value directly.

```
class Rectangle
{
private:
    double m_length = 1.0; // m_length has a default value of 1.0
    double m_width = 1.0; // m_width has a default value of 1.0

public:
    Rectangle()
    {
    // This constructor will use the default values above since they aren't overridden here
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }
};

int main()
{
    Rectangle x; // x.m_length = 1.0, x.m_width = 1.0
    x.print();

    return 0;
}
```

This program produces the result:
length: 1.0, width: 1.0

Non-static member initialization

- However, note that constructors still determine what kind of objects may be created. Consider the following case:

```
class Rectangle
{
private:
    double m_length = 1.0;
    double m_width = 1.0;

public:

    Rectangle(double length, double width)
        : m_length(length), m_width(width)
    {
        // m_length and m_width are initialized by the constructor (the default values aren't used)
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }

};

int main()
{
    Rectangle x(2.0, 3.0);
    x.print();

    return 0;
}
```

This prints:
length: 2.0, width: 3.0

Destructors

- A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed.
- Whereas constructors are designed to initialize a class, destructors are designed to help clean up.
- When an object goes out of scope normally, or a dynamically allocated object is explicitly deleted using the delete keyword, the class destructor is **automatically** called (if it exists) to do any necessary clean up before the object is removed from memory.
- Like constructors, destructors have specific naming rules:
    1) The destructor must have the same name as the class, preceded by a tilde (~).
    2) The destructor can not take arguments.
    3) The destructor has no return type.

Destructor Example

```
#include <iostream>
#include <cassert>

class IntArray
{
private:
    int *m_array;
    int m_length;

public:
    IntArray(int length) // constructor
    {
        assert(length > 0);

        m_array = new int[length];
        m_length = length;
    }

    ~IntArray() // destructor
    {
        // Dynamically delete the array we allocated earlier
        delete[] m_array ;
    }

    void setValue(int index, int value) { m_array[index] = value; }
    int getValue(int index) { return m_array[index]; }

    int getLength() { return m_length; }
};

int main()
{
    IntArray ar(10); // allocate 10 integers
    for (int count=0; count < 10; ++count)
        ar.setValue(count, count+1);

    std::cout << "The value of element 5 is: " << ar.getValue(5);

    return 0;
} // ar is destroyed here, so the ~IntArray() destructor function is called here
```

This program produces the result:

```
The value of element 5 is: 6
```

Destructor Example

- As mentioned previously, the constructor is called when an object is created, and the destructor is called when an object is destroyed. In the following example, we use cout statements inside the constructor and destructor to show this:

```
class Simple
{
private:
    int m_nID;

public:
    Simple(int nID)
    {
        std::cout << "Constructing Simple " << nID << '\n';
        m_nID = nID;
    }

    ~Simple()
    {
        std::cout << "Destructing Simple" << m_nID << '\n';
    }

    int getID() { return m_nID; }
};

int main()
{
    // Allocate a Simple on the stack
    Simple simple(1);
    std::cout << simple.getID() << '\n';

    // Allocate a Simple dynamically
    Simple *pSimple = new Simple(2);
    std::cout << pSimple->getID() << '\n';
    delete pSimple;

    return 0;
} // simple goes out of scope here
```

This program produces the following result:

```
Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1
```

5. Static Member Variables

“static” keyword

- Static duration variables are only created and initialized (to 0) once. When it goes out of scope, it is not destroyed. Each time the function generateID() is called, the value of s_value is whatever we left it at previously.

```
#include <iostream>

void incrementAndPrint()
{
    int value = 1; // automatic duration by default
    ++value;
    std::cout << value << '\n';
} // value is destroyed here

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();
}
```

```
2
2
2
```

```
#include <iostream>

void incrementAndPrint()
{
    // static duration via static keyword.
    //This line is only executed once.
    static int s_value = 1;
    ++s_value;
    std::cout << s_value << '\n';
} // s_value is not destroyed here, but becomes inaccessible

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();
}
```

```
2
3
4
```

Static member variables

```
class Something
{
public:
    int m_value = 1;
};

int main()
{
    Something first;
    Something second;

    first.m_value = 2;

    std::cout << first.m_value << '\n';
    std::cout << second.m_value << '\n';

    return 0;
}
```

```
2
1
```

- When we instantiate a class object, each object gets its own copy of all normal member variables. In this case, because we have declared two Something class objects, we end up with two copies of m_value.

Static member variables

- Member variables of a class can be made static by using the static keyword. Unlike normal member variables, static member variables are **shared** by all objects of the class.

```
class Something
{
public:
    static int s_value;
};

int Something::s_value = 1;

int main()
{
    Something first;
    Something second;

    first.s_value = 2;

    std::cout << first.s_value << '\n';
    std::cout << second.s_value << '\n';
    return 0;
}
```

```
2
2
```

Static members are not associated with class objects

- Although you can access static members through objects of the class, it is better to think of static members as belonging to the class itself, not to the objects of the class. Because s_value exists independently of any class objects, it can be accessed directly using the class name and the scope resolution operator (in this case, Something::s_value):

```
class Something
{
public:
    static int s_value; // declares the static member variable
};

int Something::s_value = 1; // defines the static member variable (we'll discuss this section below)

int main()
{
    // note: we're not instantiating any objects of type Something

    Something::s_value = 2;
    std::cout << Something::s_value << '\n';
    return 0;
}
```

- In the above snippet, s_value is referenced by class name rather than through an object. Note that we have not even instantiated an object of type Something, but we are still able to access and use Something::s_value. This is the preferred method for accessing static members.

Defining and initializing static member variables

- When we declare a static member variable inside a class, we're telling the compiler about the existence of a static member variable, but not actually defining it. Because static member variables are not part of the individual class objects, you must explicitly define the static member outside of the class, in the global scope.
- In the example above, we do so via this line:

```
int Something::s_value = 1; // defines the static member variable
```

- Note that this static member definition is not subject to access controls: you can define and initialize the value even if it's declared as private (or protected) in the class.
- If the class is defined in a .h file, the static member definition is usually placed in the associated code file for the class (e.g. Something.cpp). If the class is defined in a .cpp file, the static member definition is usually placed directly underneath the class.
- Do not put the static member definition in a header file (much like a global variable, if that header file gets included more than once, you'll end up with multiple definitions, which will cause a compile error).

An example of static member variables

- Why use static variables inside classes? One great example is to assign a unique ID to every instance of the class. Here's an example of that:

```
class Something
{
private:
    static int s_idGenerator;
    int m_id;

public:
    Something() { m_id = s_idGenerator++; } // grab the next value from the id generator

    int getID() const { return m_id; }
};

// Note that we're defining and initializing s_idGenerator even though it is declared as private above.
// This is okay since the definition isn't subject to access controls.
int Something::s_idGenerator = 1; // start our ID generator with value 1

int main()
{
    Something first;
    Something second;
    Something third;

    std::cout << first.getID() << '\n';
    std::cout << second.getID() << '\n';
    std::cout << third.getID() << '\n';
    return 0;
}
```

This program prints:

```
1
2
3
```

Static member functions

- If the static member variables are public, we can access them directly using the class name and the scope resolution operator. But what if the static member variables are private?

```
class Something
{
private:
    static int s_value;

};

int Something::s_value = 1; // initializer, this is okay even though s_value is private since it's a definition

int main()
{
    // how do we access Something::s_value since it is private?
}
```

- In this case, we can't access Something::s_value directly from main(), because it is private. Normally we access private members through public member functions.
- While we could create a normal public member function to access s_value, we'd then need to instantiate an object of the class type to use the function! We can do better. It turns out that we can also make functions static.

Static member functions

- Like static member variables, static member functions are not attached to any particular object. Here is the above example with a static member function accessor:

```
class Something
{
private:
    static int s_value;
public:
    static int getValue() { return s_value; } // static member function
};

int Something::s_value = 1; // initializer

int main()
{
    std::cout << Something::getValue() << '\n';
}
```

- Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator.
- Like static member variables, they can also be called through objects of the class type, though this is **not recommended**.

Another example

- Static member functions can also be defined outside of the class declaration. This works the same way as for normal member functions.

```
class IDGenerator
{
private:
    static int s_nextID; // Here's the declaration for a static member

public:
     static int getNextID(); // Here's the declaration for a static function
};

// Here's the definition of the static member outside the class.  Note we don't use the static keyword here.
// We'll start generating IDs at 1
int IDGenerator::s_nextID = 1;

// Here's the definition of the static function outside of the class.  Note we don't use the static keyword here.
int IDGenerator::getNextID() { return s_nextID++; }

int main()
{
    for (int count=0; count < 5; ++count)
        std::cout << "The next ID is: " << IDGenerator::getNextID() << '\n';

    return 0;
}
```

This program prints:

```
The next ID is: 1
The next ID is: 2
The next ID is: 3
The next ID is: 4
The next ID is: 5
```

- Note that because all the data and functions in this class are static, we don't need to instantiate an object of the class to make use of its functionality!

Class Definition in Head File

- All of the classes that we have written so far have been simple enough that we have been able to implement the member functions directly inside the class definition itself. E.g. below.

```
class Date
{
private:
    int m_year;
    int m_month;
    int m_day;

public:
    Date(int year, int month, int day)
    {
        setDate(year, month, day);
    }

    void setDate(int year, int month, int day)
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay()  { return m_day; }
};
```

- However, as classes get longer and more complicated, having all the member function definitions inside the class can make the class harder to manage and work with.

- Fortunately, C++ provides a way to separate the “declaration” portion of the class from the “implementation” portion. This is done by defining the class member functions outside of the class definition.

Putting class definitions in a header file

- C++ classes (and often function prototypes) are normally split up into two files. The header file has the extension of .h and contains class definitions and functions. The implementation of the class goes into the .cpp file.

File: Num.h

```
class Num
{
 private:
  int num;
 public:
  Num(int n);
  int getNum();
};
```

File: Num.cpp

```
#include "Num.h"

Num::Num() : num(0) { }
Num::Num(int n): num(n) {}
int Num::getNum()
{
 return num;
}
```

File: main.cpp

```
#include <iostream>
#include "Num.h"

using namespace std;

int main()
{
  Num n(35);
  cout << n.getNum() << endl;
  return 0;
}
```

More examples will be shown in tutorial class.

6. The hidden “this” pointer

Example

- When a member function is called, how does C++ keep track of which object it was called on?". The answer is that C++ utilizes a hidden pointer named "***this***"!

```
class Simple
{
private:
    int m_id;

public:
    Simple(int id)
    {
        setID(id);
    }

    void setID(int id) { m_id = id; }
    int getID() { return m_id; }
};
```

Here's a sample program that uses this class:

```
int main()
{
    Simple simple(1);
    simple.setID(2);
    std::cout << simple.getID() << '\n';

    return 0;
}
```

As you would expect, this program produces the result:

```
2
```

“this” pointer

- when we call *simple.setID(2);,* C++ knows that function *setID()* should operate on object simple, and that *m_id* actually refers to *simple.m_id*. Let’s examine the mechanics behind how this works.

- Take a look at the following line of code from the example :

  simple.setID(2);

- Although the call to function setID() looks like it only has one argument, it actually has two! When compiled, the compiler converts *simple.setID(2);* into the following:

  setID(&simple, 2);

“this” pointer

- Consequently, the following member function:

void setID(int id) { m_id = id; }

- is converted by the compiler into:

void setID(Simple const this, int id) { this->m_id = id; }*

- Putting it all together:
  1) When we call simple.setID(2), the compiler actually calls setID(&simple, 2).
  2) Inside setID(), the *this pointer holds the address of object simple.
  3) Any member variables inside setID() are prefixed with “this->”. So when we say m_id = id, the compiler is actually executing this->m_id = id, which in this case updates simple.m_id to id.
- The good news is that all of this happens transparently to you as a programmer, and it doesn’t really matter whether you remember how it works or not.

***this always points to the object being operated on**

Explicitly referencing *this

- Most of the time, you never need to explicitly reference the "*this*" pointer. However, there are a few occasions where doing so can be useful:
- First, if you have a constructor (or member function) that has a parameter with the same name as a member variable, you can disambiguate them by using "this":

```
class Something
{
private:
    int data;

public:
    Something(int data)
    {
        this->data = data;
    }
};
```

- Some developers prefer to explicitly add this-> to all class members. We recommend that you avoid doing so, as it tends to make your code less readable for little benefit. Using the m_ prefix is a more readable way to differentiate member variables from non-member (local) variables.

Explicitly referencing *this

- **Chaining member functions**

Method 1

```
class Calc
{
private:
    int m_value;

public:
    Calc() { m_value = 0; }

    void add(int value) { m_value += value; }
    void sub(int value) { m_value -= value; }
    void mult(int value) { m_value *= value; }

    int getValue() { return m_value; }
};
```

If you wanted to add 5, subtract 3, and multiply by 4, you'd have to do this:

```
#include <iostream>
int main()
{
    Calc calc;
    calc.add(5); // returns void
    calc.sub(3); // returns void
    calc.mult(4); // returns void

    std::cout << calc.getValue() << '\n';
    return 0;
}
```

Method 2

```
class Calc
{
private:
    int m_value;

public:
    Calc() { m_value = 0; }

    Calc& add(int value) { m_value += value; return *this; }
    Calc& sub(int value) { m_value -= value; return *this; }
    Calc& mult(int value) { m_value *= value; return *this; }

    int getValue() { return m_value; }
};
```

```
#include <iostream>
int main()
{
    Calc calc;
    calc.add(5).sub(3).mult(4);

    std::cout << calc.getValue() << '\n';
    return 0;
}
```

7. Object Relationships

- One day you're walking down the street, and you see a bright yellow object attached to a green shrubby object. You'd probably recognize that the bright yellow thing is a flower, and the green shrubby thing is a plant. Even though you'd never seen this particular type of plant before.
- How can you know all of this without ever encountering a plant of this type before?
- You know this because you understand the abstract concept of plants, and recognize that this plant is an instantiation of that abstraction.
- Life is full of recurring patterns, relationships, and hierarchies between objects.

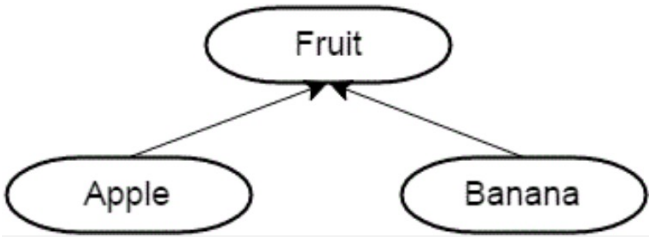
- There are many different kinds of relationships two objects may have in real-life, and we use specific “relation type” words to describe these relationships.
- For example: a square “is-a” shape. A car “has-a” steering wheel. A computer programmer “uses-a” keyboard. A flower “depends-on” a bee for pollination. A student is a “member-of” a class. And your brain exists as “part-of” you.
- All of these relation types have useful analogies in OOP.

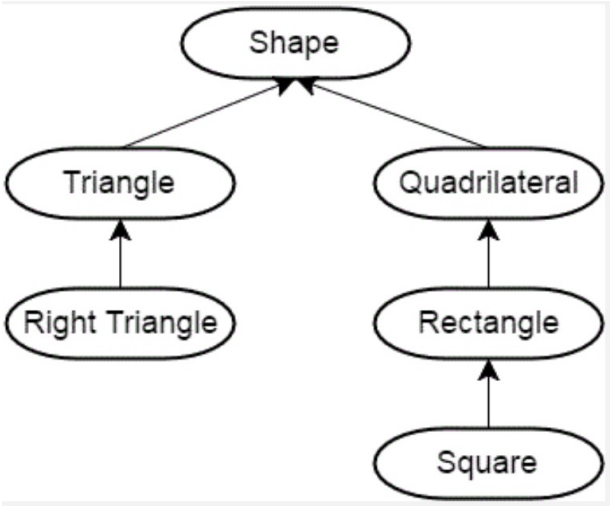
7.1 Object Relationships: Inheritance

7.1.1 Introduction to Inheritance

Introduction to Inheritance

- Consider apples and bananas. Although apples and bananas are different fruits, both have in common that they *are* fruits.
- And because apples and bananas are fruits, simple logic tells us that anything that is true of fruits is also true of apples and bananas. For example, all fruits have a name, a color, and a size. Therefore, apples and bananas also have a name, a color, and a size.
- We can say that apples and bananas inherit (acquire) these all of the properties of fruit because they *are* fruit.





Inheritance Example

- Class representing generic person:

```
#include <string>

class Person
{
public:
    std::string m_name;
    int m_age;

    Person(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};
```

A BaseballPlayer class

```
class BaseballPlayer
{
public:
    double m_battingAverage;
    int m_homeRuns;

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage(battingAverage), m_homeRuns(homeRuns)
    {
    }
};
```

Inheritance Example

- Now, we also want to keep track of a baseball player's name and age, and we already have that information as part of our Person class.
- We have three choices for how to add name and age to BaseballPlayer:

1) Add name and age to the BaseballPlayer class directly as members. This is probably the worst choice, as we're duplicating code that already exists in our Person class. Any updates to Person will have to be made in BaseballPlayer too.

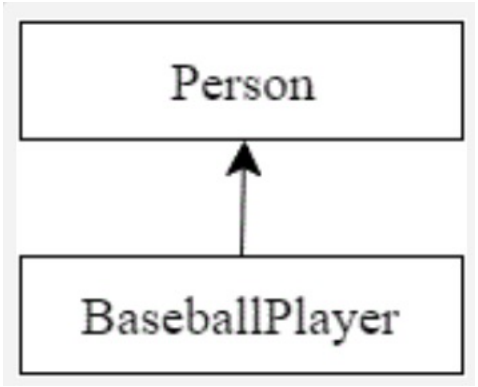
2) Add Person as a member of BaseballPlayer using composition. But we have to ask ourselves, "does a BaseballPlayer have a Person"? No, it doesn't. So this isn't the right paradigm.

3) Have BaseballPlayer inherit those attributes from Person. Remember that inheritance represents an is-a relationship. Is a BaseballPlayer a Person? Yes, it is. So inheritance is a good choice here.

Making BaseballPlayer a derived class

- After the class BaseballPlayer declaration, we use a colon, the word “public”, and the name of the class we wish to inherit. This is called *public inheritance*.

```
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage;
    int m_homeRuns;

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
        : m_battingAverage(battingAverage), m_homeRuns(homeRuns)
    {
    }
};
```



- When BaseballPlayer inherits from Person, BaseballPlayer acquires the member functions and variables from Person.

Thus, BaseballPlayer objects will have 4 member variables: m_battingAverage and m_homeRuns from BaseballPlayer, and m_name and m_age from Person.

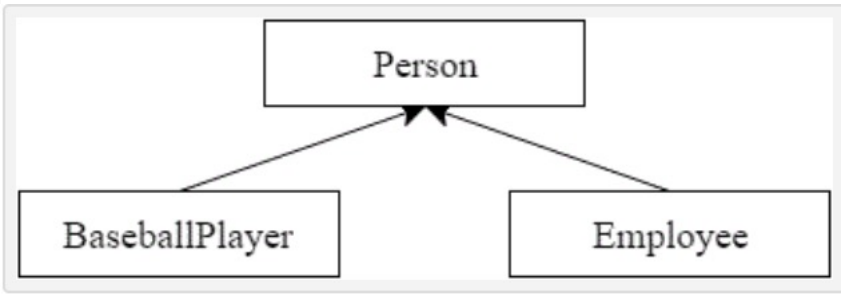
```cpp
#include <iostream>
#include <string>

class Person
{
public:
    std::string m_name;
    int m_age;

    Person(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};

// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_battingAverage;
    int m_homeRuns;

    BaseballPlayer(double battingAverage = 0.0, int homeRuns = 0)
       : m_battingAverage(battingAverage), m_homeRuns(homeRuns)
    {
    }
};

int main()
{
    // Create a new BaseballPlayer object
    BaseballPlayer joe;
    // Assign it a name (we can do this directly because m_name is public)
    joe.m_name = "Joe";
    // Print out the name
    std::cout << joe.getName() << '\n'; // use the getName() function we've acquired from the Person base class

    return 0;
}
```

Which prints the value:

```
Joe
```

An Employee derived class

```cpp
#include <iostream>
#include <string>

class Person
{
public:
    std::string m_name;
    int m_age;

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

    Person(std::string name = "", int age = 0)
        : m_name(name), m_age(age)
    {
    }
};

// Employee publicly inherits from Person
class Employee: public Person
{
public:
    double m_hourlySalary;
    long m_employeeID;

    Employee(double hourlySalary = 0.0, long employeeID = 0)
        : m_hourlySalary(hourlySalary), m_employeeID(employeeID)
    {
    }

    void printNameAndSalary() const
    {
        std::cout << m_name << ": " << m_hourlySalary << '\n';
    }
};

int main()
{
    Employee frank(20.25, 12345);
    frank.m_name = "Frank"; // we can do this because m_name is public

    frank.printNameAndSalary();

    return 0;
}
```



This prints:

```
Frank: 20.25
```

7.1.2 Order of construction of derived classes

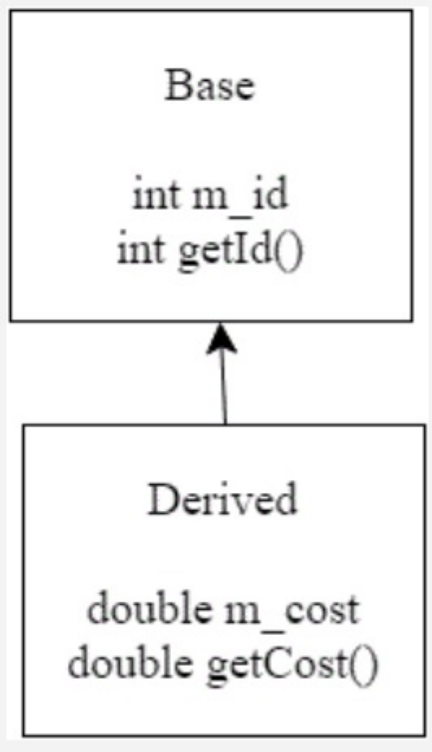
Order of construction of derived classes

- Let us take a closer look at the order of construction that happens when a derived class is instantiated.

```
class Base
{
public:
    int m_id;

    Base(int id=0)
        : m_id(id)
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0)
        : m_cost(cost)
    {
    }

    double getCost() const { return m_cost; }
};
```

- In this example, class Derived is derived from class Base.

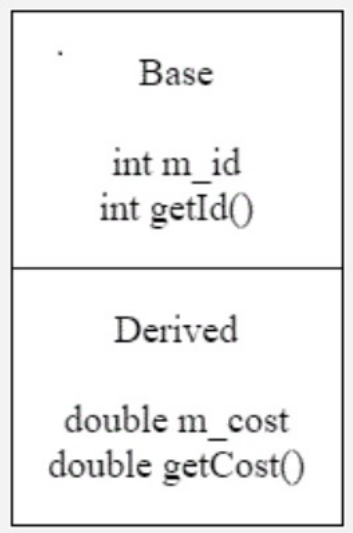
Order of construction of derived classes



Because Derived inherits functions and variables from Base, you may assume that the members of Base are copied into Derived.

Order of construction of derived classes

- However, this is not true. Instead, we can consider Derived as a two part class: one part Derived, and one part Base.



Order of construction of derived classes

```
int main()
{
    Derived derived;

    return 0;
}
```

- As mentioned above, Derived is really two parts: a Base part, and a Derived part. When C++ constructs derived objects, it does so in phases. First, the **most-base** class (at the top of the inheritance tree) is constructed first. Then **each child class** is constructed in order, **until the most-child class** (at the bottom of the inheritance tree) is constructed last.

- So when we instantiate an instance of Derived, first the Base portion of Derived is constructed (using the Base default constructor). Once the Base portion is finished, the Derived portion is constructed (using the Derived default constructor).

Order of construction of derived classes

```
#include <iostream>

class Base
{
public:
    int m_id;

    Base(int id=0)
        : m_id(id)
    {
        std::cout << "Base\n";
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0)
        : m_cost(cost)
    {
        std::cout << "Derived\n";
    }

    double getCost() const { return m_cost; }
};

int main()
{
    std::cout << "Instantiating Base\n";
    Base cBase;

    std::cout << "Instantiating Derived\n";
    Derived cDerived;

    return 0;
}
```

This program produces the following result:

```
Instantiating Base
Base
Instantiating Derived
Base
Derived
```

Order of construction for inheritance chains

- It is sometimes the case that classes are derived from other classes, which are themselves derived from other classes.

```
class A
{
public:
    A()
    {
        std::cout << "A\n";
    }
};

class B: public A
{
public:
    B()
    {
        std::cout << "B\n";
    }
};

class C: public B
{
public:
    C()
    {
        std::cout << "C\n";
    }
};

class D: public C
{
public:
    D()
    {
        std::cout << "D\n";
    }
};
```

```
int main()
{
    std::cout << "Constructing A: \n";
    A cA;

    std::cout << "Constructing B: \n";
    B cB;

    std::cout << "Constructing C: \n";
    C cC;

    std::cout << "Constructing D: \n";
    D cD;
}
```

```
Constructing A:
A
Constructing B:
A
B
Constructing C:
A
B
C
Constructing D:
A
B
C
D
```

Conclusion

- C++ constructs derived classes in phases, starting with the most-base class (at the top of the inheritance tree) and finishing with the most-child class (at the bottom of the inheritance tree). As each class is constructed, the appropriate constructor from that class is called to initialize that part of the class.

7.1.3 Constructors and initialization of derived classes

Constructors and initialization of derived classes

- we've explored some basics around inheritance in C++ and the order that derived classes are initialized. In this lesson, we'll take a closer look at the role of constructors in the initialization of derived classes.

```
class Base
{
public:
    int m_id;

    Base(int id=0)
        : m_id(id)
    {
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0)
        : m_cost(cost)
    {
    }

    double getCost() const { return m_cost; }
};
```

- With non-derived classes, constructors only have to worry about their own members. For example, consider Base. We can create a Base object like this:

```
int main()
{
    Base base(5); // use Base(int) constructor

    return 0;
}
```

Constructors and initialization of derived classes

- Here's what actually happens when base is instantiated:
  1. Memory for base is set aside
  2. The appropriate Base constructor is called
  3. The initialization list initializes variables
  4. The body of the constructor executes
  5. Control is returned to the caller

Constructors and initialization of derived classes

- This is pretty straightforward. With derived classes, things are slightly more complex:

```
int main()
{
    Derived derived(1.3); // use Derived(double) constructor

    return 0;
}
```

- Here's what actually happens when derived is instantiated:
    1. Memory for derived is set aside (enough for both the Base and Derived portions)
    2. The appropriate Derived constructor is called
    3. **The Base object is constructed first using the appropriate Base constructor**. If no base constructor is specified, the default constructor will be used.
    4. The initialization list initializes variables
    5. The body of the constructor executes
    6. Control is returned to the caller

Initializing base class members

- One of the current shortcomings of our Derived class as written is that there is no way to initialize m_id when we create a Derived object (we don't want m_id always be the defalut 0). What if we want to set both m_cost (from the Derived portion of the object) and m_id (from the Base portion of the object) when we create a Derived object?
- Fortunately, C++ gives us the ability to explicitly choose which Base class constructor will be called! To do this, simply add a call to the base class Constructor in the initialization list of the derived class:

```
class Derived: public Base
{
public:
    double m_cost;

    Derived(double cost=0.0, int id=0)
        : Base(id), // Call Base(int) constructor with value id!
          m_cost(cost)
    {
    }

    double getCost() const { return m_cost; }
};
```

```
int main()
{
    Derived derived(1.3, 5); // use Derived(double, int) constructor
    std::cout << "Id: " << derived.getId() << '\n';
    std::cout << "Cost: " << derived.getCost() << '\n';

    return 0;
}
```

The base class constructor Base(int) will be used to initialize m_id to 5, and the derived class constructor will be used to initialize m_cost to 1.3!

Thus, the program will print:

```
Id: 5
Cost: 1.3
```

Constructors and initialization of derived classes

- In more detail, here's what happens:

1. Memory for derived is allocated.
2. The Derived(double, int) constructor is called, where cost = 1.3, and id = 5
3. The compiler looks to see if we've asked for a particular Base class constructor. We have! So it calls Base(int) with id = 5.
4. The base class constructor initialization list sets m_id to 5
5. The base class constructor body executes, which does nothing
6. The base class constructor returns
7. The derived class constructor initialization list sets m_cost to 1.3
8. The derived class constructor body executes, which does nothing
9. The derived class constructor returns

Now we can make our members private

- Now that you know how to initialize base class members, there's no need to keep our member variables public. We make our members variables private again, as they should be.

```
class Base
{
private: // our member is now private
    int m_id;

public:
    Base(int id=0)
        : m_id(id)
    {
    }

    int getId() const { return m_id; }
};
class Derived: public Base
{
private: // our member is now private
    double m_cost;

public:
    Derived(double cost=0.0, int id=0)
        : Base(id), // Call Base(int) constructor with value id!
          m_cost(cost)
    {
    }

    double getCost() const { return m_cost; }
};

int main()
{
    Derived derived(1.3, 5); // use Derived(double, int) constructor
    std::cout << "Id: " << derived.getId() << '\n';
    std::cout << "Cost: " << derived.getCost() << '\n';

    return 0;
}
```

In the above code, we've made m_id and m_cost private. This is fine, since we use the relevant constructors to initialize them, and use a public accessor to get the values. This prints, as expected:

```
Id: 5
Cost: 1.3
```

Another example

```
#include <string>
class Person
{
private:
    std::string m_name;
    int m_age;

public:
    Person(std::string name = "", int age = 0)
        : m_name(name), m_age(age )
    {
    }

    std::string getName() const { return m_name; }
    int getAge() const { return m_age; }

};
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
private:
    double m_battingAverage;
    int m_homeRuns;

public:
    BaseballPlayer(std::string name = "", int age = 0,
        double battingAverage = 0.0, int homeRuns = 0)
        : Person(name, age), // call Person(std::string, int) to initialize these fields
            m_battingAverage(battingAverage), m_homeRuns(homeRuns)
    {
    }

    double getBattingAverage() const { return m_battingAverage; }
    int getHomeRuns() const { return m_homeRuns; }
};
```

Now we can create baseball players like this:

```
int main()
{
    BaseballPlayer pedro("Pedro Cerrano", 32, 0.342, 42);

    std::cout << pedro.getName() << '\n';
    std::cout << pedro.getAge() << '\n';
    std::cout << pedro.getHomeRuns() << '\n';

    return 0;
}
```

This outputs:

```
Pedro Cerrano
32
42
```

Inheritance chains

- Classes in an inheritance chain work in exactly the same way.

```
#include <iostream>

class A
{
public:
    A(int a)
    {
        std::cout << "A: " << a << '\n';
    }
};

class B: public A
{
public:
    B(int a, double b)
    : A(a)
    {
        std::cout << "B: " << b << '\n';
    }
};

class C: public B
{
public:
    C(int a , double b , char c)
    : B(a, b)
    {
        std::cout << "C: " << c << '\n';
    }
};

int main()
{
    C c(5, 4.3, 'R');

    return 0;
}
```

this program prints:

```
A: 5
B: 4.3
C: R
```

In this example, class C is derived from class B, which is derived from class A.

Summary

- When constructing a derived class, the derived class constructor is responsible for determining which base class constructor is called. If no base class constructor is specified, the default base class constructor will be used. In that case, if no default base class constructor can be found (or created by default), the compiler will display an error. The classes are then constructed in order from most base to most derived.

7.1.4 Inheritance and access specifiers

The protected access specifier

- Besides **public** and **private**, C++ has a third access specifier that we have yet to talk about because it's only useful in an inheritance context.
- The **protected** access specifier allows the class the member belongs to and derived classes to access the member. However, protected members are not accessible from outside the class.

```
class Base
{
public:
    int m_public; // can be accessed by anybody
private:
    int m_private; // can only be accessed by Base members and friends (but not derived classes)
protected:
    int m_protected; // can be accessed by Base members, friends, and derived classes
};

class Derived: public Base
{
public:
    Derived()
    {
        m_public = 1; // allowed: can access public base members from derived class
        m_private = 2; // not allowed: can not access private base members from derived class
        m_protected = 3; // allowed: can access protected base members from derived class
    }
};

int main()
{
    Base base;
    base.m_public = 1; // allowed: can access public members from outside class
    base.m_private = 2; // not allowed: can not access private members from outside class
    base.m_protected = 3; // not allowed: can not access protected members from outside class
}
```

Different kinds of inheritance, and their impact on access

- There are three different ways for classes to inherit from other classes: public, private, and protected.
-

```
// Inherit from Base publicly
class Pub: public Base
{
};

// Inherit from Base privately
class Pri: private Base
{
};

// Inherit from Base protectedly
class Pro: protected Base
{
};

class Def: Base // Defaults to private inheritance
{
};
```

- If you do not choose an inheritance type, C++ defaults to private inheritance (just like members default to private access if you do not specify otherwise).

Public inheritance

Access specifier in base class	Access specifier when inherited publicly
Public	Public
Private	Inaccessible
Protected	Protected

```
class Base
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Pub: public Base // note: public inheritance
{
    // Public inheritance means:
    // Public inherited members stay public (so m_public is treated as public)
    // Protected inherited members stay protected (so m_protected is treated as protected)
    // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
    Pub()
    {
        m_public = 1; // okay: m_public was inherited as public
        m_private = 2; // not okay: m_private is inaccessible from derived class
        m_protected = 3; // okay: m_protected was inherited as protected
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_private = 2; // not okay: m_private is private in Base
    base.m_protected = 3; // not okay: m_protected is protected in Base

    Pub pub;
    pub.m_public = 1; // okay: m_public is public in Pub
    pub.m_private = 2; // not okay: m_private is inaccessible in Pub
    pub.m_protected = 3; // not okay: m_protected is protected in Pub
```

Private inheritance

Access specifier in base class	Access specifier when inherited privately
Public	Private
Private	Inaccessible
Protected	Private

```
class Base
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};

class Pri: private Base // note: private inheritance
{
    // Private inheritance means:
    // Public inherited members become private (so m_public is treated as private)
    // Protected inherited members become private (so m_protected is treated as private)
    // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
    Pri()
    {
        m_public = 1; // okay: m_public is now private in Pri
        m_private = 2; // not okay: derived classes can't access private members in the base class
        m_protected = 3; // okay: m_protected is now private in Pri
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    // In this case, the access specifiers of base.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_private = 2; // not okay: m_private is private in Base
    base.m_protected = 3; // not okay: m_protected is protected in Base

    Pri pri;
    pri.m_public = 1; // not okay: m_public is now private in Pri
    pri.m_private = 2; // not okay: m_private is inaccessible in Pri
    pri.m_protected = 3; // not okay: m_protected is now private in Pri
```

Protected inheritance

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Private	Inaccessible
Protected	Protected

Example

```
class Base
{
public:
    int m_public;
private:
    int m_private;
protected:
    int m_protected;
};
```

```
class D2 : private Base // note: private inheritance
{
    // Private inheritance means:
    // Public inherited members become private
    // Protected inherited members become private
    // Private inherited members stay inaccessible
public:
    int m_public2;
private:
    int m_private2;
protected:
    int m_protected2;
};
```

D2 can access its own members without restriction. D2 can access Base's m_public and m_protected members, but not m_private. Because D2 inherited Base privately, m_public and m_protected are now considered private when accessed through D2. This means the public can not access these variables when using a D2 object, nor can any classes derived from D2.

```
class D3 : public D2
{
    // Public inheritance means:
    // Public inherited members stay public
    // Protected inherited members stay protected
    // Private inherited members stay inaccessible
public:
    int m_public3;
private:
    int m_private3;
protected:
    int m_protected3;
};
```

D3 can access its own members without restriction. D3 can access D2's m_public2 and m_protected2 members, but not m_private2. Because D3 inherited D2 publicly, m_public2 and m_protected2 keep their access specifiers when accessed through D3. D3 has no access to Base's m_private, which was already private in Base. Nor does it have access to Base's m_protected or m_public, both of which became private when D2 inherited them.

Summary

- Here's a table of all of the access specifier and inheritance types combinations:

Access specifier in base class	Access specifier when inherited publicly	Access specifier when inherited privately	Access specifier when inherited protectedly
Public	Public	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible
Protected	Protected	Private	Protected

7.1.5 Calling inherited functions and overriding behavior

Calling a base class function

- When a member function is called with a derived class object, the compiler first looks to see if that member exists in the derived class. If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the parent classes. It uses the first one it finds.

```
class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value(value)
    {
    }

    void identify() { std::cout << "I am a Base\n"; }
};

class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }
};
int main()
{
    Base base(5);
    base.identify();

    Derived derived(7);
    derived.identify();

    return 0;
}
```

This prints

```
I am a Base
I am a Base
```

Redefining behaviors

- When derived.identify() is called, the compiler looks to see if function identify() has been defined in the Derived class. It hasn't. Then it starts looking in the inherited classes (which in this case is Base). Base has defined an identify() function, so it uses that one. In other words, Base::identify() was used because Derived::identify() doesn't exist.
- However, if we had defined Derived::identify() in the Derived class, it would have been used instead.

```
class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }

    int getValue() { return m_value; }

    // Here's our modified function
    void identify() { std::cout << "I am a Derived\n"; }
};
```

```
int main()
{
    Base base(5);
    base.identify();

    Derived derived(7);
    derived.identify();

    return 0;
}
```

```
I am a Base
I am a Derived
```

Redefining behaviors

- Note that when you redefine a function in the derived class, the derived function does not inherit the access specifier of the function with the same name in the base class. It uses whatever access specifier it is defined under in the derived class. Therefore, a function that is defined as private in the base class can be redefined as public in the derived class, or vice-versa!

```
class Base
{
private:
    void print()
    {
        std::cout << "Base";
    }
};

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

};


int main()
{
    Derived derived;
    derived.print(); // calls derived::print(), which is public
    return 0;
}
```

Adding to existing functionality

- Sometimes we don't want to completely replace a base class function, but instead want to add additional functionality to it. In the above example, note that Derived::identify() completely hides Base::identify()! This may not be what we want. It is possible to have our derived function call the base version of the function of the same name (in order to reuse code) and then add additional functionality to it.

```
class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }

    int GetValue() { return m_value; }

    void identify()
    {
        Base::identify(); // call Base::identify() first
        std::cout << "I am a Derived\n"; // then identify ourselves
    }
};
```

```
int main()
{
    Base base(5);
    base.identify();

    Derived derived(7);
    derived.identify();

    return 0;
}
```

```
I am a Base
I am a Base
I am a Derived
```

Adding to existing functionality

- When derived.identify() is executed, it resolves to Derived::identify(). However, the first thing Derived::identify() does is call Base::identify(), which prints “I am a Base”. When Base::identify() returns, Derived::identify() continues executing and prints “I am a Derived”.

```
class Derived: public Base
{
public:
    Derived(int value)
        : Base(value)
    {
    }

    int GetValue() { return m_value; }

    void identify()
    {
        identify(); // Note: no scope resolution!
        cout << "I am a Derived";
    }
};
```

Function “identity” will be called **Recursively**.

7.2 Compsition Relation

Composition Relation

- In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts.
- This process of building complex objects from simpler ones is called **object composition**.
- To qualify as a **composition**, an object and a part must have the following relationship:
  - The part (member) is part of the object (class)
  - The part (member) can only belong to one object (class) at a time
  - The part (member) has its existence managed by the object (class)
  - The part (member) does not know about the existence of the object (class)
- A good real-life example of a composition is the relationship between a person's body and a heart.
- the part doesn't know about the existence of the whole. Your heart operates blissfully unaware that it is part of a larger structure. We call this a **unidirectional** relationship, because the body knows about the heart, but not the other way around.

Composition Example

```
class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction(int numerator=0, int denominator=1):
        m_numerator(numerator), m_denominator(denominator)
    {
        // We put reduce() in the constructor to ensure any fractions we make get reduced!
        // Since all of the overloaded operators create new Fractions, we can guarantee this will get called here
        reduce();
    }
};
```

This class has two data members: a numerator and a denominator. The numerator and denominator are part of the Fraction (contained within it). They can not belong to more than one Fraction at a time. The numerator and denominator don't know they are part of a Fraction, they just hold integers. When a Fraction instance is created, the numerator and denominator are created.

Implementing compositions

- They are implemented as normal data members. Because these data members exist directly as part of the class, their lifetimes are bound to that of the class instance itself.

```cpp
#ifndef POINT2D_H
#define POINT2D_H

#include <iostream>

class Point2D
{
private:
    int m_x;
    int m_y;

public:
    // A default constructor
    Point2D()
        : m_x(0), m_y(0)
    {
    }

    // A specific constructor
    Point2D(int x, int y)
        : m_x(x), m_y(y)
    {
    }

    void display()
    {
        std::cout << "now at: " << m_x << "," << m_y << std::endl;
    }

    // Access functions
    void setPoint(int x, int y)
    {
        m_x = x;
        m_y = y;
    }

};
```

```cpp
#ifndef CREATURE_H
#define CREATURE_H

#include <iostream>
#include <string>
#include "Point2D.h"

class Creature
{
private:
    std::string m_name;
    Point2D m_location;

public:
    Creature(const std::string &name, const Point2D &location)
        : m_name(name), m_location(location)
    {
    }

    void display()
    {
        m_location.display();
    }

    void moveTo(int x, int y)
    {
        m_location.setPoint(x, y);
    }
};

#endif /* CREATURE_H */
```

Implementing compositions

```
#include <string>
#include <iostream>
#include "Creature.h"
#include "Point2D.h"

int main()
{
    std::cout << "Enter a name for your creature: ";
    std::string name;
    std::cin >> name;
    Creature creature(name, Point2D(4, 7));

    while (1)
    {
        // print the creature's name and location
        creature.display();

        std::cout << "Enter new X location for creature (-1 to quit): ";
        int x=0;
        std::cin >> x;
        if (x == -1)
            break;

        std::cout << "Enter new Y location for creature (-1 to quit): ";
        int y=0;
        std::cin >> y;
        if (y == -1)
            break;

        creature.moveTo(x, y);
        }

    return 0;
}
```

```
Enter a name for your creature: Bob
now at: 4,7
Enter new X location for creature (-1 to quit): 3
Enter new Y location for creature (-1 to quit): 2
now at: 3,2
Enter new X location for creature (-1 to quit): -1
```

Aggregation

- To qualify as an **aggregation**, a whole object and its parts must have the following relationship:
  - The part (member) is part of the object (class)
  - The part (member) can belong to more than one object (class) at a time
  - The part (member) does *not* have its existence managed by the object (class)
  - The part (member) does not know about the existence of the object (class)
- Like a composition, an aggregation is still a part-whole relationship, where the parts are contained within the whole, and it is a unidirectional relationship.
- However, unlike a composition, parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts.

Aggregation

- For example, consider the relationship between a person and their home address.
- Address can belong to more than one person at a time: for example, to both you and your roommate. Address isn't managed by the person -- the address probably existed before the person got there, and will exist after the person is gone. Additionally, a person knows what address they live at, but the addresses don't know what people live there.

Implementing Aggregations

- In an aggregation, we also add parts as member variables. However, these member variables are typically either references or pointers that are used to point at objects that have been created outside the scope of the class.
- Consequently, an aggregation usually either takes the objects it is going to point to as constructor parameters, or it begins empty and the subobjects are added later via access functions.
- Because these parts exist outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed (but not deleted). Consequently, the parts themselves will still exist.

Aggregation Example

```
#include <string>
#include <iostream>

class Teacher
{
private:
    std::string m_name;

public:
    Teacher(std::string name)
        : m_name(name)
    {
    }

    std::string getName() { return m_name; }
};

class Department
{
private:
    Teacher *m_teacher; // This dept holds only one teacher for simplicity, but it could hold many teachers

public:
    Department(Teacher *teacher = nullptr)
        : m_teacher(teacher)
    {
    }
};

int main()
{
    // Create a teacher outside the scope of the Department
    Teacher *teacher = new Teacher("Bob"); // create a teacher
    {
        // Create a department and use the constructor parameter to pass
        // the teacher to it.
        Department dept(teacher);

    } // dept goes out of scope here and is destroyed

    // Teacher still exists here because dept did not delete m_teacher

    std::cout << teacher->getName() << " still exists!";

    delete teacher;

    return 0;
}
```

We're going to make a couple of simplifications:

First, the department will only hold one teacher.

Second, the teacher will be unaware of what department they're part of.

Pick the right relationship for what you're modeling

- Although it might seem a little silly in the above example that the Teacher's don't know what Department they're working for, that may be totally fine in the context of a given program. When you're determining what kind of relationship to implement, implement the simplest relationship that meets your needs, not the one that seems like it would fit best in a real-life context.
- For example, if you're writing a body shop simulator, you may want to implement a car and engine as an aggregation, so the engine can be removed and put on a shelf somewhere for later. However, if you're writing a racing simulation, you may want to implement a car and an engine as a composition, since the engine will never exist outside of the car in that context.
- *Rule: Implement the simplest relationship type that meets the needs of your program, not what seems right in real-life.*

Summarizing composition and aggregation

- Compositions:
  - Typically use normal member variables
  - Can use pointer members if the class handles object allocation/deallocation itself
  - Responsible for creation/destruction of parts

- Aggregations:
  - Typically use pointer or reference members that point to or reference objects that live outside the scope of the aggregate class
  - Not responsible for creating/destroying parts

Composition and Aggregation can co-exist

- It is worth noting that the concepts of composition and aggregation are not mutually exclusive, and can be mixed freely within the same class. It is entirely possible to write a class that is responsible for the creation/destruction of some parts but not others.
- For example, our Department class could have a name and a Teacher. The name would probably be added to the Department by composition, and would be created and destroyed with the Department. On the other hand, the Teacher would be added to the department by aggregation, and created/destroyed independently.

Association Example

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// Since these classes have a circular dependency, we're going to forward declare Doctor
class Doctor;

class Patient
{
private:
        std::string m_name;
        vector<Doctor *> m_doctor; // so that we can use it here

        // We're going to make addDoctor private because we don't want the public to use it.
        // They should use Doctor::addPatient() instead, which is publicly exposed
        // We'll define this function after we define what a Doctor is
        // Since we need Doctor to be defined in order to actually use anything from it


public:
        Patient(string name): m_name(name){}

        void addDoctor(Doctor *doc);
        // We'll implement this function below Doctor since we need Doctor to be defined at that point
        void display_doctors();

        string getName() const { return m_name; }
};
```

Association Example

```
class Doctor
{
private:
        string m_name;
        vector<Patient *> m_patient;

public:
        Doctor(string name):
            m_name(name)
        {
        }

        void addPatient(Patient *pat)
        {
            // Our doctor will add this patient
            m_patient.push_back(pat);

            // and the patient will also add this doctor
            pat->addDoctor(this);
        }

        void display_patients(){
            unsigned int length = m_patient.size();
            if (length == 0){
                cout << m_name << " has no patients right now";
            }
            cout << m_name << " is seeing patients: ";
            for (unsigned int count = 0; count < length; ++count)
                cout << m_patient[count]->getName() << ' ';
        }
        const string getName()  { return m_name; }
};

void Patient::addDoctor(Doctor *doc)
{
        m_doctor.push_back(doc);
}

void Patient::display_doctors(){
    unsigned int length = m_doctor.size();
    if (length == 0){
        cout << m_name << " has no doctors right now";
    }
    cout << m_name << " is seeing doctors: ";
    for (unsigned int count = 0; count < length; ++count)
        cout << m_doctor[count]->getName() << ' ';
}
```

Association Example

```
int main()
{
        // Create a Patient outside the scope of the Doctor
        Patient *p1 = new Patient("Dave");
        Patient *p2 = new Patient("Frank");
        Patient *p3 = new Patient("Betsy");

        Doctor *d1 = new Doctor("James");
        Doctor *d2 = new Doctor("Scott");

        d1->addPatient(p1);

        d2->addPatient(p1);
        d2->addPatient(p3);

        (*d1).display_patients(); cout << '\n';
        (*d2).display_patients(); cout << '\n';
        (*p1).display_doctors(); cout << '\n';
        (*p2).display_doctors(); cout << '\n';
        (*p3).display_doctors(); cout << '\n';

        delete p1;
        delete p2;
        delete p3;

        delete d1;
        delete d2;

        return 0;
}
```

```
James is seeing patients: Dave
Scott is seeing patients: Dave Betsy
Dave is seeing doctors: James Scott
Frank has no doctors right nowFrank is seeing doctors:
Betsy is seeing doctors: Scott
```