Classes

Class Declaration

```
class MyClass {
/* public, protected and private variables,
constants, and functions */
};
```

Class Declaration

```
class A
{
private:
      int x, y;
      char theChar;
public:
      Foo();
      ~Foo();
      void doCoolStuff();
      int get_x() { return x; }
      int get_y() { return y; }
      char get_theChar() { return theChar; }
};
```

Class Members

A class can have the following kinds of members

1. **Data members**
   a. non-static data members, including bit fields.
   b. static data members
2. **member functions**
   a. non-static member functions
   b. static member functions

Class Members

3. **nested types**
   a. ***nested classes*** and ***enumerations*** defined within the class definition
   b. aliases of existing types, defined with *typedef* declarations
   c. the name of the class within its own definition acts as a public member type alias of itself for the purpose of lookup (except when used to name a constructor): this is known as *injected-class-name*

https://stackoverflow.com/questions/4571355/why-would-one-use-nested-classes-in-c
http://www.learncpp.com/cpp-tutorial/8-15-nested-types-in-classes/

Class Members

4. ***enumerators*** from all unscoped enumerations defined within the class
5. ***member templates*** (class templates or function templates) may appear in the body of any non-local class/struct/union

http://en.cppreference.com/w/cpp/language/member_template

Class Members

```
/* start of Enclosing class declaration */
class Enclosing {
    int x;
    /* start of Nested class declaration */
    class Nested {
        int y;
        void NestedFun(Enclosing *e) {
          cout<<e->x;  // works fine: nested class can access
                       // private members of Enclosing class
        }
    }; // declaration Nested class ends here
}; // declaration Enclosing class ends here
```

Class Members

```
class List {
    public:
        List(): head(NULL), tail(NULL) {}
    private:
        class Node {
            public:
                int data;
                Node* next;
                Node* prev;
        };
    private:
        Node* head;
        Node* tail;
};
```

Class Members

```
void X() { }
class X {
public:
   static X create() { return X(); }
};
```

The injected class name means that X is declared as a member of X, so that name lookup inside X always finds the current class, not another X that might be declared at the same enclosing scope

https://stackoverflow.com/questions/25549652/why-is-there-an-injected-class-name

Access Labels

Public

can be accessed freely anywhere a declared object is in scope

Private

Members (member variables and helper functions) only accessible within the class defining them, or friend classes

Protected

protected members are accessible in the class that defines them and in classes that inherit from that base class, or friends of it

Class Inheritance

- **Deriving** one class from another class
- **Polymorphism** allows redefining how member functions of the same name operate
- **Base class** (parent class, or superclass): initial class used as basis for derived class
- **Derived class** (child class, or subclass): new class that incorporates all of the data and member functions of its base class
  - Usually adds new data and function members
  - Can override any base class function

Class Inheritance

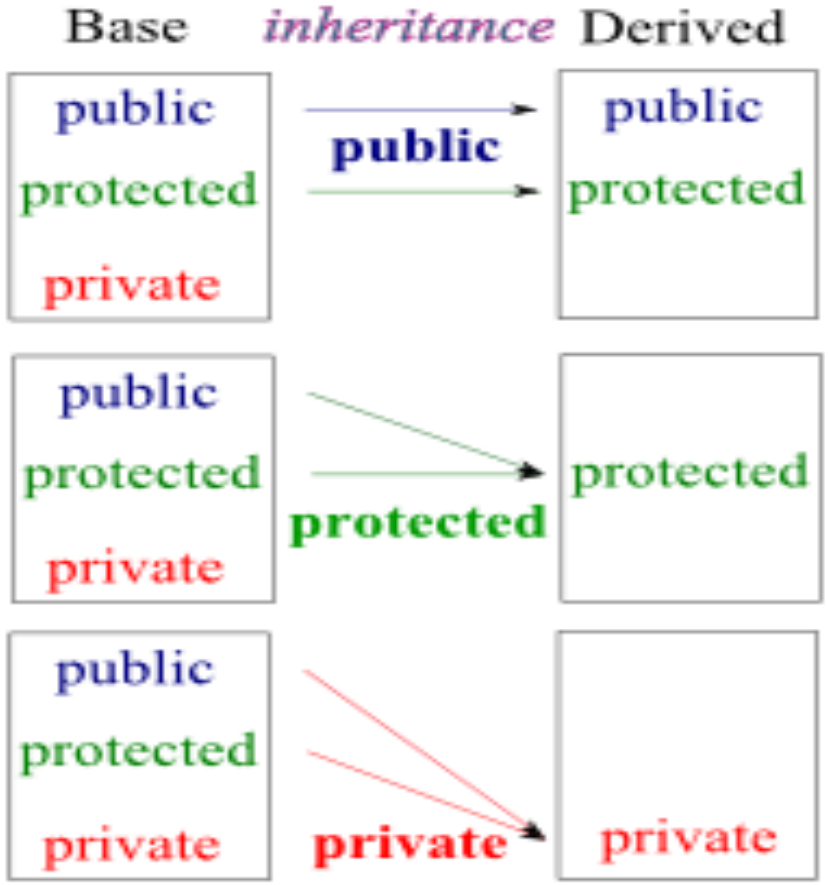
There are three types of class inheritance: public, private and protected

	Private	protected	public
private inheritance	The member is inaccessible	The member is private	The member is private
protected inheritance	The member is inaccessible	The member is protected	The member is protected
public inheritance	The member is inaccessible	The member is protected	The member is public

https://en.wikibooks.org/wiki/C%2B%2B_Programming/Classes#Inheritance_.28Derivation.29

Class Inheritance

There are three types of class inheritance: public, private and protected



http://www.bogotobogo.com/cplusplus/private_inheritance.php

Class Inheritance

- Protected members are inherited as protected methods in public inheritance
- Use the protected label when want to declare a method inaccessible outside the class and not to lose access to it in derived classes
- Losing accessibility can be useful
  - encapsulating details in the base class.

Class Inheritance

```
class Person {};
class Student :private Person {};// private
void eat(const Person& p) {}      // anyone can eat
void study(const Student& s) {}  // only students
                                  // study

int main()
{
    Person p; // p is a Person
    Student s;// s is a Student
    eat(p);   // fine, p is a Person
    eat(s);   // error! s isn't a Person
    return 0;
}
```

Multiple inheritance

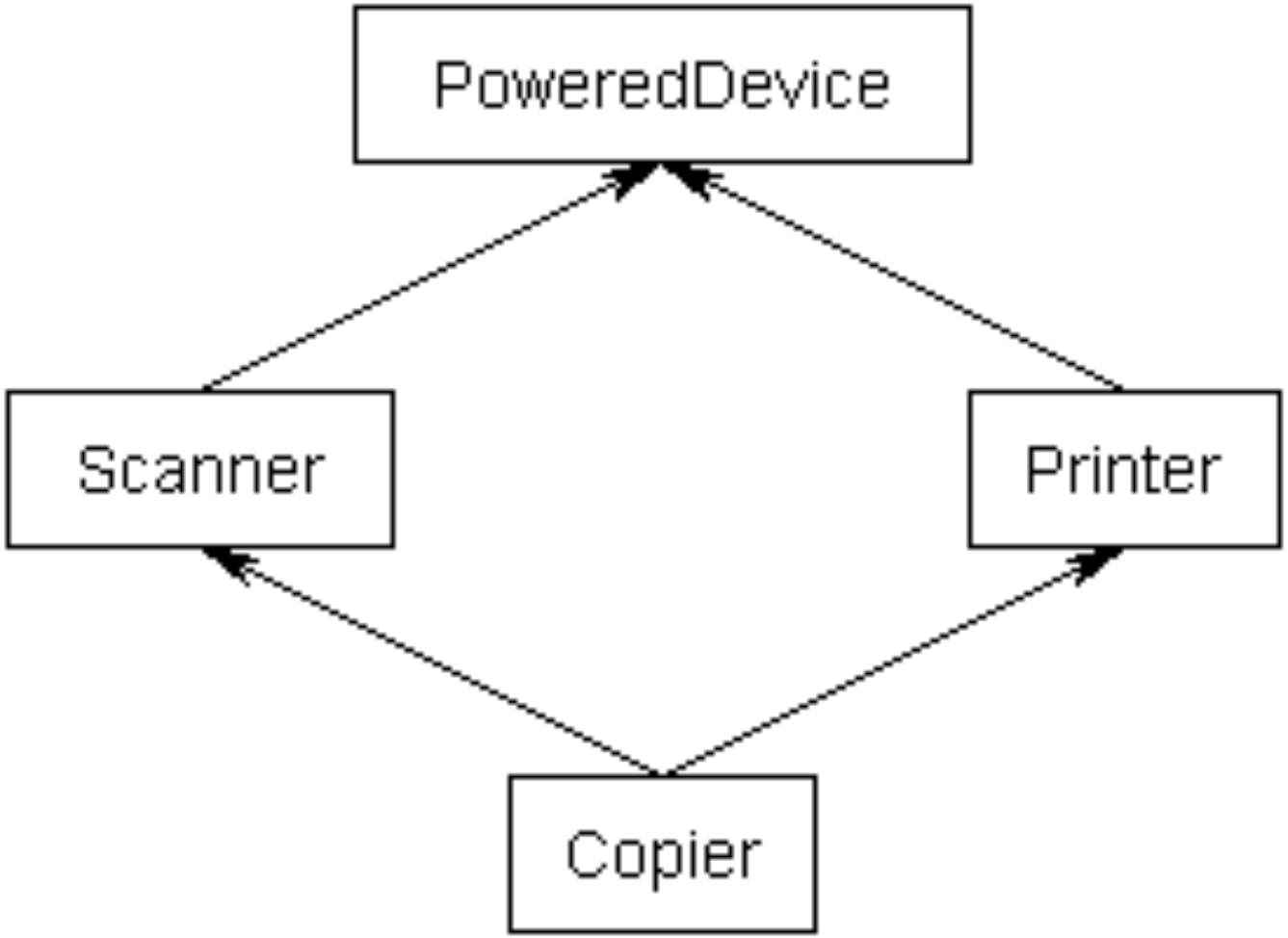
- Inheritance describes a relationship between two classes in which one class (the *child* class) *subclasses* the *parent* class.
- The child inherits methods and attributes of the parent, allowing for shared functionality
- Multiple inheritance allows programmers to use more than one totally orthogonal hierarchy simultaneously

https://en.wikipedia.org/wiki/Multiple_inheritance

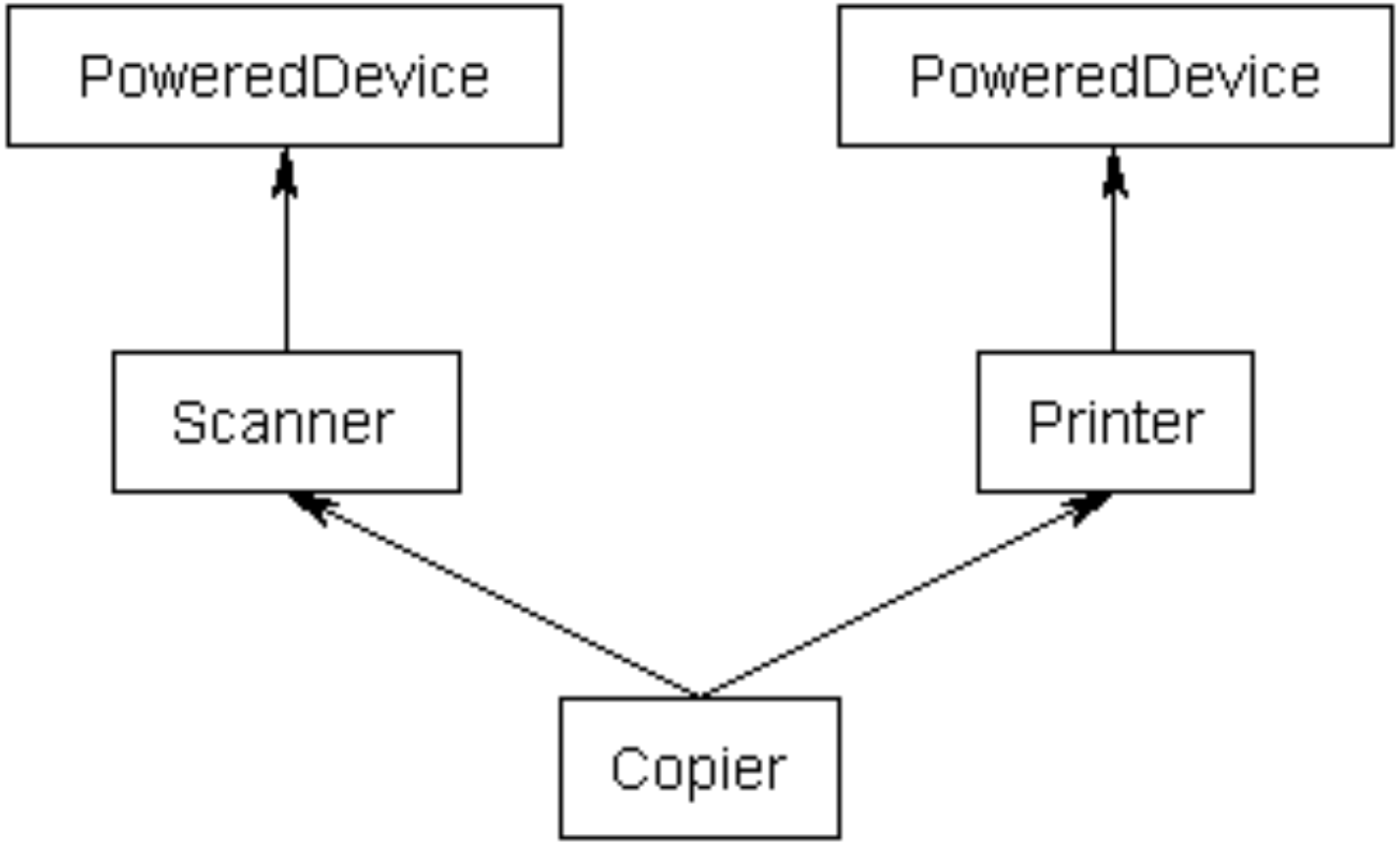
Multiple inheritance

```
class PoweredDevice
{
};
class Scanner: public PoweredDevice
{
};
class Printer: public PoweredDevice
{
};
class Copier: public Scanner, public Printer
{
};
```

Multiple inheritance



Multiple inheritance



Multiple inheritance

```
class PoweredDevice
{
};
class Scanner: virtual public PoweredDevice
{
};
class Printer: virtual public PoweredDevice
{
};
class Copier: public Scanner, public Printer
{
};
```

Multiple inheritance

1. Virtual base classes are created before non-virtual base classes, which ensures all bases get created before their derived classes
2. The `Scanner` and `Printer` constructors still have calls to the `PoweredDevice` constructor
   - Creating an instance of `Copier`, these constructor calls are simply ignored because `Copier` is responsible for creating the `PoweredDevice` , not `Scanner` or `Printer`
   - Creating an instance of `Scanner` or `Printer` , the virtual keyword is ignored, those constructor calls would be used, and normal inheritance rules apply

More on Virtual inheritance:
http://www.cprogramming.com/tutorial/virtual_inheritance.html

Multiple inheritance

3. If a class inherits one or more classes that have virtual parents, the most derived class is responsible for constructing the virtual base class
   - In this case, **`Copier`** inherits **`Printer`** and **`Scanner`**, both of which have a **`PoweredDevice`** virtual base class
   - **`Copier`**, the most derived class, is responsible for creation of **`PoweredDevice`**
   - Note that this is true even in a single inheritance case: if **`Copier`** was singly inherited from **`Printer`**, and **`Printer`** was virtually inherited from **`PoweredDevice`**, **`Copier`** is still responsible for creating **`PoweredDevice`**

http://www.learncpp.com/cpp-tutorial/118-virtual-base-classes/

Abstract Class

- An **abstract class** is a **class** that is designed to be specifically used as a base **class**
- An **abstract class** contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (`= 0`) in the declaration of a virtual member function in the **class** declaration
- A pure virtual function is one which **must be overridden** by any concrete (i.e., non-abstract) derived class

Abstract Class

```
class AbstractClass
{
  public:
      virtual void AbstractMemberFunction() = 0;
      // Pure virtual function makes
      // this class Abstract class.
      virtual void NonAbstractMemberFunction1();
      // Virtual function.
  void NonAbstractMemberFunction2();
};
```

Data Members

`this` pointer

- The `this` keyword acts as a pointer to the class being referenced
- The `this` pointer is only accessible within nonstatic member functions of a `class`, `union` or `struct`, and is not available in static member functions

static data member

- shared by all instances of the owner class and derived classes
- static member variables are not part of the individual class objects
- must explicitly define the static member outside of the class

Members Functions

Define the function outside of the class definition using the scope resolution operator "::“

Functions within classes can access and modify (unless the function is constant) data members without declaring them, because the data members are already declared in the class

Members Functions

Overloading

Multiple member functions can exist with the same name on the same scope, but must have different signatures

A member function's signature is comprised of the member function's name and the type and order of the member function's parameters

Constructors and other class member functions, except the Destructor, can be overloaded

Encapsulation

- The process of keeping the details about how an object is implemented hidden away from users
- Implement encapsulation via access specifiers

- ✓ **encapsulated classes help protect your data and prevent misuse**
- ✓ **encapsulated classes are easier to change**
- ✓ **encapsulated classes are easier to debug**

Constructors

Member function that is automatically called when an object of that class is instantiated

Name:

- Constructors should always have the same name as the class (with the same capitalization)
- Constructors have no return type (not even void)

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

Constructors

constructors with parameters

- Multiple constructors can be defined for a class
  - Each must be distinguishable by the number and types of its parameters
  - If no constructor function is written, compiler assigns default constructor
- The general format of a constructor method:

```
className::className(parameter list)
{
    // method body
}
```

Constructors

- If your class has no other constructors, C++ will automatically create an empty default constructor
- If you do have other non-default constructors in your class, but no default constructor, C++ will not create an empty default constructor
- In this case, the class will not be instantiatable without parameters

Constructors

Constructor initialization lists

```
class Something
{
private:
    int m_value1;
    double m_value2;
    char m_value3;
public:
    Something()
    { // These are all assignments
        m_value1 = 1;
        m_value2 = 2.2;
        m_value3 = 'c';
    }
};
```

Constructors

Constructor initialization lists

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

```
public:
    Something(int value1, double value2, char value3='c')
        : m_value1(value1), m_value2(value2), m_value3(value3)
```

Constructors

Uniform initialization in C++11

```
class Something
{
private:
    const int m_value;

public:
    Something(): m_value { 5 } // Uniformly initialize
    {
    }
};
```

Destructors

- Same name as the Class preceded with a "~“
- It can not have arguments and can't be overloaded
- The Destructor is invoked when:
  - Objects are destroyed
  - after the function they were declared in returns,
  - when the **delete** operator is used
  - when the program is over

`static` member function

- Member functions or variables declared static are shared between all instances of an object type
  - only one copy of the member function or variable does exists for any object type
- A static function does not operate on a specific instance and thus does not take a "`this`" pointer (behaving like a free function)
  - static class functions can be called without creating instances of the class

`static` member function

```
class Foo {
public:
     Foo() {
       ++numFoos;
       cout << "Created " << numFoos
            << " instances of the Foo class\n";
     }
     static int getNumFoos() {
       return numFoos;
     }
private:
     static int numFoos;
};
int Foo::numFoos = 0; // allocate memory and initialize
int main() {
      Foo f1; Foo f2; Foo f3;
      cout << "we've made " << Foo::getNumFoos()
           << " instances of the Foo class\n";
}
```

Named Constructors

- ***Named constructors*** is the name given to functions used to create an object of a class without (directly) using its constructors
- This might be used for the following:
  - To circumvent the restriction that constructors can be overloaded only if their signatures differ.
  - Making the class non-inheritable by making the constructors private.
  - Preventing stack allocation by making constructors private

https://en.wikibooks.org/wiki/C%2B%2B_Programming/Classes#Inheritance_.28Derivation.29

`const` member function

```
Circle copy(Circle&) const;
```

- A `const` member function is indicated by a `const` suffix just after the member function's parameter list
- It cannot call any non-const member functions, nor can it change any member variables
- The function can only be called via a `const` object of the class.
- Must be member function of the class `'Circle'`.

```
Circle copy(const Circle &);
```

- This one means that the parameter passed cannot be changed within the function. This one may or may not be a member function

Accessors and Modifiers

Accessor (Getter)

An accessor is a member function that does not modify the state of an object. The accessor functions should be declared as **const**

Modifier (Setter)

A member function that changes the value of at least one data member. In other words, an operation that modifies the state of an object. Modifiers are also known as ‘mutators’

Polymorphism

- Permits same function name to invoke one response in objects of base class and another response in objects of derived class
  - Example of polymorphism: overriding of base member function using an overloaded derived member function
- **Function binding**: determines whether base class or derived class version of function will be used
  - **Static binding**: determination of which function to be called is made at compile time
    - Used in normal function calls
  - **Dynamic binding**: determination of which function to be called is made at runtime (via virtual function)

Polymorphism

- **Virtual function :** creates pointer to function to be used
  - Value of pointer variable is not established until function is actually called
  - At runtime, and on the basis of the object making the call, the appropriate function address is used