1. Inheritance in C++

1.1 Why use inheritance?

- The most important aspect of inheritance is not that it provides member functions for the new class.
  It's the relationship expressed between the new class and the base class. Saying, "The new class *is a type of* the existing class", summarizes this relationship.

- Inheritance and composition support *incremental development* by allowing you to introduce new code without causing bugs in existing code.

- Code reuse

- Inheritance is an essential feature of an object-oriented programming language. As is data abstraction and polymorphism.

1.2 Inheritance syntax

```
class X
{
  public:
    X() { i = 0; }
    void set(int ii) { i = ii; }
    int read() const { return i; }
    int permute() { return i = i * 47; }
  private:
      int i;

};

class Y : public X
{
  public:
    Y() { i = 0; }
    int change()   //A completely new operation
    {
      i = permute(); //Call (base) class operation
      return i;
    }
    void set(int ii) //set() is redefined
    {
      i = ii;
      X::set(ii); //call base class operation set()
    }
  private:
    int i; // Different from X's i
};
```

- **Y** contains a subobject of **X** just as if you had created a member object of **X** inside **Y** instead of inheriting from X.
- The **private** elements of **X** are still there, they take up space – you just can't access them directly.
- During inheritance, everything defaults to **privat**e.

1.3 The constructor initializer list

- When an object is created, the compiler guarantees that constructors for all of its subobjects (super classes) are called.

- What happens if your subobjects (super classes) don't have default constructors, or if you want to change a default argument in a constructor?

  - The new class constructor doesn't have permission to access the **private** data elements of the subobject (super class), so it can't initialize them directly.

  - The solution is simple: Call the constructor for the subobject (super class).

For a class **MyTyp**e, inherited from **Ba**r, this might look like this:

```
MyType::MyType(int i) : Bar(i) { // ...
```

  - if **Bar** has a constructor that takes a single **int** argument.

1.4 Order of constructor and destructor calls

- Construction starts at the very root of the class hierarchy.

- At each level the base class constructor is called first, followed by the member object constructors.

- The destructors are called in exactly the reverse order of the constructors

- It's also interesting that the order of constructor calls for member objects is completely unaffected by the order of the calls in the constructor initializer list.

The order is determined by the order that the member objects are declared in the class. If you could change the order of constructor calls via the constructor initializer list, you could have two different call sequences in two different constructors. But the poor destructor wouldn't know how to properly reverse the order of the calls for destruction, and you could end up with a dependency problem.

1.5 Name hiding

If you inherit a class and provide a new definition for one of its member functions, there are two possibilities.

- The first is that you provide the exact signature and return type in the derived class definition as in the base class definition. This is called *redefining* for ordinary member functions and *overriding* when the base class member function is a **virtual** function **(virtual** functions are the normal case, and will be covered later).

- The second possibility is that you can change the member function argument list and/or the return type in the derived class.

Remark: Notice that the only time you can talk about redefinition of functions is during inheritance; with a member object you can only manipulate the public interface of the object, not redefine it.

What will be the effect? Here is an example:

```
class Base
{
  public:
    int f() const
    {
      cout << "Base::f()\n";
      return 1;
    }
    int f(string) const { return 1; }
    void g() {}
};

class Derived1 : public Base
{
  public:
    //Redefining
    void g() {}
};

class Derived2 : public Base
{
  public:
  // Redefinition: the second overloaded form of f is now unavailable
  int f() const
  {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base
{
  public:
  // Change return type: hides both base-class versions
  void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base
{
  public:
  // Change argument list: hides both base-class versions
  int f(int) const
  {
    cout << "Derived4::f()\n";
    return 4;
  }
};
```

- In general, we can say that anytime you redefine an overloaded function name from the base class, all the other versions are automatically hidden in the new class.
- The addition of the **virtual** keyword affects function overloading a bit more, which we will see later.

1.5.1 A word of warning

If you change the interface of the base class by modifying the signature and/or return type of a member function from the base class, then you're using the class in a different way than inheritance is normally intended to support.
It doesn't necessarily mean you're doing it wrong, it's just that the ultimate goal of inheritance is to support *polymorphis*m, and if you change the function signature or return type then you are actually changing the interface of the base class. If this is what you have intended to do then you are using inheritance primarily to reuse code, and not to maintain the common interface of the base class (which is an essential aspect of polymorphism).
In general, when you use inheritance this way it means you're taking a general-purpose class and specializing it for a particular need – which is usually, but not always, considered the realm of composition.

- However inheritance in an **object-oriented sense** should be about an "is a" relationship. If you feel that you have to change the interface of a given base-class then the relationship isn't really an "is a" one. If you're not creating a subclass that *is-a* type of the baseclass, then why are you inheriting?
  If you are changing the interface of a baseclass when programming in an object-oriented paradigm you are most likely to be experiencing a flaw in design. **Do some more thinking!**

1.5.2 Inheritance and static member functions

Static member functions act the same as non-**static** member functions:

1. They inherit into the derived class.

2. If you redefine a static member, all the other overloaded functions in the base class are hidden.

However, **static** member functions cannot be **virtual,** this is covered later

1.6 Functions that are not inherited

- Constructors and destructors are not inherited and must be created specially for each derived class.

- In addition, the **operator=** is not inherited because it performs a constructor-like activity.

1.7 Composition vs. inheritance

- When should one be chosen over the other?

- Typically, you use composition to reuse existing types as part of the underlying implementation of the new type

- Inheritance is used when you want to force the new type to be the same type as the base class (type equivalence guarantees interface equivalence). Since the derived class has the base-class interface, it can be *upcast* to the base, which is critical for polymorphism

- Composition is generally used when you want the features of an existing class inside your new class, but not its interface.

- The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.

A simple example would be trying to model a car. A car has an engine, doors, windows and so on. So modeling the car composed of these objects should be fine. It would however not be very wise to claim that a car is composed of a vehicle, a car **is** a vehicle. This relation should therefor be modeled using inheritance.

1.8 Private inheritance

In inheritance, all classes inherited defaults to private, for example:

```
class Y
{ public:
    void func1(){ cout << "Y::func1()"; }
    //. . .
  private:
    int func2(){ return 1;}
    //. . .
};
class X : Y{ //. . . };   <==>   class X : private Y {//. . . };
```

- The meaning is that everything inherited from Y becomes private in X. In X, Y is only part of the underlying implementation.
- The class user has no access to the underlying functionality, and an object of X cannot be treated as an instance of the base class, Y.
- **private** inheritance is included in the language for "completeness", but if for no other reason than to reduce confusion, you'll usually want to use composition rather than **private** inheritance.
- Private inheritance can be used to produce part of the same interface as the base class *and* disallow the treatment of the object as if it were a base-class object.
- If you want any private members to be visible, just declare their names (no arguments or return values) in the **public** section of the derived class:

```
class X : private Y
{
  public:
    Y::func2;
//. . .

};
```

Warning: Using **private** inheritance to hide part of the functionality of the base class is a very specialized scenario. If you are programming in an object-oriented fashion you generally don't want to do this because this breaks the paradigm.
Another issue is combining private inheritance with runtime type identification (RTTI), this is related to particular complications. The interested student is encouraged to explore this subject on his/her own.

1.9 Protected

It is with inheritance that the keyword **protected** has a meaning. Anything declared as **protected** says:

“This is **private** as far as the class user is concerned, but available to anyone who inherits from this class.”

- Ideally there would be no use for protected members, since this breaks encapsulation.
- Practically however, some people claim that it has a use. Some also say, "it needs to be in the language for completeness". Though one could argue that completeness is most likely itself an ideal, which no language yet has managed to implement. Paradoxical, isn't it?
- If you are going to use **protected** access*, always leave the data members* ***private***. If you don't, you are in trouble because you can't change the underlying implementation of the baseclass.

1.9.1 Protected inheritance

Using protected inheritance as in:

```
class X : protected Y {//. . . };
```

means “implemented-in-terms-of” to other classes but “is-a” for derived classes and friends. It’s something you don’t use very often, but it’s in the language for "completeness".

1.10 Operator overloading & inheritance

- Except for the assignment operator, operators are automatically inherited into a derived class.

Remark: The reason for the assignment operator not to be inherited automatically is because of its constructor-like behavior. The meaning of an assignment in a subclass can differ substantially from the assignment in its baseclass.

1.11 Upcasting

```
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument
{
  public:
    void play(note) const { cout << "Instrument::play()"; }
};

class Wind : public Instrument
{
  public:
    void play(note) const { cout << "Wind::play()"; }

};

void tune(Instrument& i)
{
  // ...
  i.play(middleC);

}

void main()
{
  Wind flute;
  tune(flute); // Upcasting
}
```

- Allowed automatically by the compiler because it is safe. The only thing that can occur to the class interface is that it can lose member functions, not gain them.

1.12 Upcasting and the copy-constructor

- If you allow the compiler to synthesize a copy-constructor for a derived class, it will automatically call the base-class copy-constructor, and then the copy-constructors for all the member objects (or perform a bitcopy on built-in types) so you'll get the right behavior:

- However, if you write your own copy-constructor and you make the mistake not to call the baseclass copy-constructor explicitly, the default constructor for the baseclass will be called. This causes the baseclass portion of the object not to be initialized properly.

- Therefore, **always remember to properly call the base-class copy-constructor whenever you write your own copy-constructor.**

1.13 Pointer and reference upcasting

```
Wind w;
Instrument* ip = &w; // Upcast, no explicit cast needed
Instrument& ir = w; // Upcast, no explicit cast needed
```

There is a problem however, what will happen below?

```
ip->play(middleC);
```

Of course, the base-class version of play() will be called, this is an undesirable behavior which is solved by polymorphism.