Inheritance

Inheritance should be used when

1. some group of types have something in common,
2. you want to able to mix the types at run time,
3. You don't know in advance how many types there will be. (You want to be able to add more types later.) Put differently: Adding another type would not mess up all your algorithms.

Examples of Inheritance

I have used inheritance only a few times:

1. Different scenery objects in a flight-simulation program.
2. Defining a common interface to simulatable objects in flight simulation program (same as above).

Inheritance Should not be Used:

- When the types do not occur simultaneously. In that case, consider using a template.
- When you have group of subtypes that together define one type. For example, in logic one has propositional operators $\bot, \top, \neg, \wedge, \vee, \rightarrow, \leftrightarrow$ and quantifiers $\forall, \exists$.

  Creating different subtypes for different types of formulas would spread all logical algorithms (for example substitution or rewriting) over the different subtypes. It would make the code unreadable.
- When you want a function to return values of different type. Use either a union type, or exceptions.

Inheritance

If type T_1 inherits from T_2, then T_1 is a subtype of T_2.

T_1 must have all methods that T_1 has, with the same signature.

In addition, the methods must have the same meaning and preserve the same invariants.

Interface

An interface is a is a type that other types inherit from, but which has no elements by itself.

monarchy $\subseteq$ **country**, **republic** $\subseteq$ **country**.

Do there exist countries that are not monarchies or republics? That is a difficult design choice. If you make the wrong choice, this may cause a lot of problems later.

In C^{++}, interfaces are represented by abstract classes.

Virtual vs. Non-Virtual Methods

- If `method()` is not declared **virtual** in class `T`, the compiler always calls `T :: method( )` on objects of type `T`.
- If `method( )` is declared **virtual** in `T`, compiler creates code that looks at the object at runtime. If the object has a type `U` that inherits from `T`, and `U` also has `method( )`, then `U::method( )` will be called.

  It is important that you understand that **(1)** the decision is made at run time, **(2)** the most specialized method is called, **(3)** unfortunately, this works only for pointers and references.

Virtual Methods

Suppose T4 inherits from T3 inherits from T2 inherits from T1.
Suppose T3 and T1 have a definition of the method.

```
T4* t4;
t4 -> method( );  // Calls method of t3.

T3* t3;
t3 -> method( );  // Calls method of t3.

T2* t2;
t2 -> method( );  // Calls method of t1.

T1* t1;
t1 -> method( );  // Calls method of t1.
```

The example on the previous page was artificial. In reality, long chains of inheritance should be avoided.

Linker Errors

If somewhere in the tree of inheritance, a virtual method is not defined, the result will be a linker error: Linker errors are usually unreadable.

It is possible to specify that the top class `T` does not have a given method. In that case, it is not possible to create elements of `class T` and `T` is called an interface or abstract class. If `class T` does not have a given method, then the method is called pure in `T`.

```
class T
{
   void method( ) = 0;
      // Specifies that method( ) is pure in T.
};
```

Derived Classes and Private Members

Derived classes cannot touch private members. (Otherwise, the concept of **private** member would become meaningless.)

A **protected** member can be seen by derived classes but by nobody else.

Dynamic Cast

A dynamic cast can cast a derived class to a base class.

It has two posible forms:

```
const T2* t = dynamic_cast< T2 * > ( t );
const T2* t = dynamic_cast< const T2* > ( t );
   // Result is 0 if types don't fit.

T2& t = dynamic_cast< T2& > (t);
const T2& t = dynamic_cast< const T2& >(t);
   // Throws an exception when types don't fit.
   // Should be used only if you know that t is
   // of proper type.
```

Don't use C-style casts!

Run Time Selection for Non-Member Methods

Run time resolution is only possible for member functions.

This means that binary operators and functions that are not members (like `<<`) are problematic.

In the case of +, the best solution is to define a virtual method `print`, and to make `operator <<` call the `print` method in the base class.

Dealing with <<

Assuming that `tt` is the base class, from which all other classes inherit:

```
std::ostream& operator << ( std::ostream& stream,
                            const tt& t )
{
   t. print( stream );
      // Print must be virtual method of tt.
}
```

Access to Fields of Base Class

Assume that we have

```
class aa
{
   int x;
};

class bb : public aa
{
   int y;
};
```

In the methods of `bb`, the field `x` can be accessed. It is also possible to write `b.x` for an object of type `bb`.

Constructors

Assume that we have

```
class aa
{
   int x;
};

class bb : public aa
{
   int y;
};
```

The field `x` can be accessed in the methods of `bb`, but not in an initializer.

Constructors (2)

Every object of type `bb` contains an object of type `aa`, which has to be initialized by explicitly calling its constructor.

```
bb::bb( int x, int y )
   : aa(x),
     y(y)
{ }
```

Writing a direct initializer `x(x)` is not allowed, because `bb` is not a friend of `aa`. Otherwise `bb` could bypass the constructors of `aa` and break its invariants.

How does it all Work?

Suppose that `a` is of class `aa`. Supppose that `method( )`is virtual in `aa`.

Every object that belongs to a class that inherits from `aa` has a tag (the Run Time Type Information, RTTI) that marks to which class the object belongs.

Method `method( )` of class `aa` contains a table, called the vtable. Using this table, `method( )` decides at run time, using the RTTI, which derived `method( )` should be called.

Making this decision costs some small amount of time. This is the reason why the `virtual` keyword exists.

For each method, the vtable can be filled in, only at link time, because the compiler cannot know all classes that inherit from a base class. This is the reason why failing to provide a method in a subclass causes linker errors.

Writing `= 0` behind a method declaration tells the compiler that the corresponding entry in the vtable should remain empty, so that the linker will not complain about it. This makes the class abstract.

Printing the Type Information

```
#include <typeinfo>

std::cout << typeid( *p ). name( ); // For pointers.
std::cout << typeid( r ). name( );  // For references.
```

Prints a string that shows the type. Use this for debugging only!

Do not use **`typeid`** for method selection! For this, you should use only polymorphism and `dynamic_cast< >`.

The difference with `dynamic_cast< >` is that `dynamic_cast< >` also accepts derived classes.

Destructors

In case the desctructor does something non-trivial, it has to be declared virtual in the base class.

It is easy to declare virtual destructors, but one should not forget to do it.

Restoring Object Semantics

In C^{++}, polymorphism is possible only for references and for pointers.

This causes problems if you want to fill a container with polymorphic objects.

Consider for example a computer algebra system, where you have different types of number classes, (for example **int**, **real**, maybe also symbolic numbers), and you want to put all these types of numbers in a vector.

Restoring Object Semantics (2)

Another example is the FS program, where there are different scenery objects (points, lines, surfaces, polyhedra) that all need to be drawn.

If you put pointers in a vector or a list, copy constructor, assignment, and destructor, will not work well. (Expect crashes and memory leaks.)

Restoring Object Semantics (3)

Solution:

Write an additional class, which contains a pointer, and which has the constructor/copying/assignment.

Put this additional class in the container.

Disadvantage is that you have to invent another class name, but it is the only solution that I know.

Restoring Object Semantics (4)

Let's do the example with numbers. Suppose we have **my_real**, **my_rational**, **my_integer**. All these need to inherit from **num**. This class **num** should not be used by the user.

Restoring Object Semantics (5)

```
class my_real : public num
{
   ...
};

class my_rational : public num
{
   ...
};

class my_integer : public num
{
   ...
};
```

Restoring Object Semantics (6)

```
class number
{
   num* ref;
      // Makes the polymorphism possible.

   number( const number& n );
   void operator = ( const number& n );
   ~number( );
      // Restore object semantics.
};
```

Recommended Approach

Make sure that each derived class has a `clone( ) const` method:

```
my_real* my_real::clone( ) const
{
   return new my_real( *this );
}

my_rational* my_rational::clone( ) const
{
   return new my_rational( *this );
}


...
```

If necessary, make sure that each derived class has a destructor.

Recommended Approach (2)

Also make sure that each derived class has a `print( std::ostream& stream ) const` method:

```
void my_real::print( std::ostream& stream ) const
{
   ...
}

void my_rational::print( std::ostream& stream ) const
{
   ...
}

...
```

Recommended Approach (3)

The `clone( ) const` and the `print( std::ostrea& stream ) const` method must be virtual and abstract in the helper class `num`.

```
class num
{
   virtual num* clone( ) const = 0;
   virtual void print( std::ostream& ) = 0;
};
```

Recommended Approach: Constructors

```
class number
{
   // From derived class:
   number( const num& n )
      : ref( n. clone( ))
   {
   }

   // Copy Constructor:
   number( const number& n )
      : ref( n. ref -> clone( ))
   {
   }
```

Recommended Approach: Assignment

```
void operator = ( const num& n )
{
    if( &n != ref )
    {
        delete ref;
        ref = n. clone( );
    }
}
```

Recommended Approach: Assignment (2)

```
void operator = ( const number& n )
{
   if( n. ref != ref )
   {
      delete ref;
      ref = n. ref. clone( );
   }
}
```

Recommended Approach: Desctructors

If required, each of the derived classes should have a desctructor, and **`num`** must have a virtual destructor.

Class number must have:

```
number:: ~number( )
{
   delete ref;
}
```

Recommended Approach: Getting the Contents

In class **number**, one can decide to hide the **num**. This is possible if class **number** has sufficiently many methods to do all the necessary operations.

Otherwise, the user of **number** must be able to access the **num**. This can be done by adding a **getcontents( )** method as follows:

```
const num* number::getcontents( ) const { return ref; }
num* number::getcontents( ) const { return ref; }
```

Recommended Approach: Printing

```
std::ostream& number::operator << ( std::ostream& stream,
                                    const number& n )
{
   n. getcontents( ) -> print( stream );
      // Or n. ref -> print( stream );
      // if we are friend of number.
   return stream;
}
```

In order to define binary operators that select at run time, use either dynamic_cast, or create a tree of member functions.