Inheritance and Subclasses

In C^{++} it is possible to declare a class T_1 as subclass of another class T_2. This can be done if:

1. Every T_1 is a T_2.
2. Every method of T_2 has a good, non-artificial specification that can be fulfilled in a natural way by a method of T_1.

The reals are not a subclass of the integers.

In the I/O Stream library, **ostringstream** and **ofstream** are subclasses of **ostream**.

Subclasses are usually grouped in trees:

1. Some group of types have something important in common,
2. You want to able to mix these types at run time. There must be variables or elements in containers, of which you cannot predict in advance (at the latest at compile time) which of the type in the group will be there.

Examples of Subclasses: Modelling Real World objects

Subclasses naturally occur when real world objects are modelled:

1. Different types of scenery objects in a flight-simulation program. They inherit from a common base **scenery_object**.

   Also, one can define a subclass **beacon** from which navigation aids (VOR,NBD,ILS) inherit.

2. Different types of files in the C^{++} standard library are grouped as subclasses. Files are almost physical objects.

Use of Inheritance in Implementing Inductive Types

Inheritance can be used as implementation technique in the definition of inductive types.

- A **bignum** is an expression.
- A **variable** is an expression.
- If $e_1, \ldots, e_n$ are expressions, f is a function name, then $f(e_1, \ldots, e_n)$ is also an expression.

The three types can be defined as subclasses of **class expr**. If you use this subclasses in this way, you should make it invisible from outside **class expr**. It is better to define a helper class.

Subclasses should not be Used:

- When the classes never mix at run time. In that case, use templates.
- When the types involved are not physical in nature.

Inheritance versus Polymorphism

Containers in the STL are polymorphic. For each variable of a container type, it decided at compile time which type of objects they will hold.

Polymorphic structures can be completely typechecked at compile time.

Subclasses are needed when a variable must be able to hold objects of different types at different times.

Type correctness can be checked only partially at compile time.

Declaring Subclasses in C^{++}.

```
struct electricdevice
{
   std::string brand;  // Brand (Siemens, Toshiba, etc.)
   double price;     // estimated.
   double life_expectation; // estimated in years.
};

struct computer : public electricdevice
{
   std::string processor;
   double clockfrequencey;     // in Mhz.
   unsigned int memory;        // in MB.
   unsigned int harddisksize;  // in MB.
};
```

```
struct washingmachine : public electricdevice
{
   double sizeofdrum; // Maximum weight that
                      // can be washed at once.
   double rpm;
};

struct laptop : public computer
{
   double screensize; //  in inches.
   double batterylife; // In hours.
   bool matt;     // True if screen is matt.
};
```

public means that the subclass relation is publicly visible.

Subclasses in C^{++}

If B is a subclass of A,

- Every B contains an A as subobject.
- Conversion from $B\star$ to $A\star$ is possible. Every method of A is also a method of B.
- Conversion from $B\&$ to $A\&$ is possible.

Only References and Pointers are Subtypes

A B itself cannot be converted into an A!

The compiler needs to know the size of every local variable, field in a struct, or element of a container.

Since the compiler cannot know if A has subclasses, and which size these subclasses have, it cannot reserve a space that is big enough to hold a B.

Hence there is no way that a B can fit into the space of an A.

(Only the linker sees the complete class hierarchy.)

If you try to assign a B to an A, the B will be sliced.

Constructors

In the constructor, the baseclass object is constructed by explicitly calling its constructor:

```
washingmachine( const std::string& brand,
                double price, double life_expectation,
                double sizeofdrum, double rpm )
   : electricdevice( brand, price, life_expectation ),
     sizeofdrum{ sizeofdrum }, rpm{ rpm }
{ }
```

electricdevice must have a constructor that fits. If it is not initialized, it will be default initialized. In that case, there must exist a default constructor.

Constructors (2)

```
washingmachine( double sizeofdrum, double rpm,
                const electricdevice& e )
   : electricdevice{ e },
     sizeofdrum{ sizeofdrum },
     rpm{ rpm }
{ }
```

Inheritance of Fields and Methods

A **washingmachine** contains an **electricdevice**.

It has all fields of **electricdevice**. These are **brand**, **price**, and **life_expectation**.

Also, every method of **electricdevice** can be called with a **washingmachine**.

For example, one can define

```
double electricdevice::costperyear( ) const
   { return price / life_expectation; }
```

and call it with a **washingmaschine**. This is called inheritance (of fields and methods.)

The complete base class object is not available as field even though it is there. **washingmachine** does not have a field **electricdevice**.

Virtual Functions

In the example before, function **costperyear** is defined for **electricmachine**. It can see only the fields of **electricmachine**.

Very often, this is not what we want:

Suppose that we want to write a function **printspecification** that prints the main specifications of the **electricdevice**.

A **computer** has more to tell than an **electricdevice**, and it would be pity to miss this extra information.

Switching is not an Option

We don't want to write such code:

```
void printspecification( std::ostream& out,
                         const electricdevice& e )
{
   if( e is a washingmaschine )
   {
      std::cout << "drum size " << e.sizeofdrum << "\n";
      std::cout << "rotations per minute "<< e.rpm << "\n";
      etc.
   }
   if( e is a computer )
      ...
   if( e is a television )
      ...
}
```

What is bad about this?

We want knowledge about the details of the class to be in the class, not at some other place.

When a new type of electricdevice is added, we don't want to search for all possible places where code depends on the type of the electronicdevice.

Virtual Functions

Solution: Define function **printspecification** as a virtual member function of **electric device**.

A derived class can either inherit the method from its base class, or override it.

When **e. printspecfication(out)** is called, the run time evironment looks at the exact type of **e**, and calls the proper version of **printspecification( )**.

Virtual Functions

In file **electricdevice.h** write:

```
virtual void printspecification( std::ostream& out ) const
   // Keyword ‘virtual’ means that we allow run time
   // selection of printspecification( ).
```

In file **electricdevice.cpp**:

```
void electricdevice::printspecification(
                              std::ostream& out ) const
{
   out << "brand " << brand << "\n";
   out << "price " << price << "\n";
   out << "life expectation " << life_expectation << "\n";
}
```

In file **washingmachine.cpp**:

```
void washingmachine::printspecification(
            std::ostream& out ) const override
{
   out << "washing machine:\n";
   out << "drum size " << sizeofdrum << "\n";
   out << "rotations per minute " << rpm << "\n";

   // If you want, you can print the fields of the
   // electronic device separately:
   out << ...

   // or you call:
   electricdevice::printspecification( out );
}
```

Use of Virtual Functions

```
std::ostream& operator << ( std::ostream& stream,
                            const electricdevice& e )
{
   e. printspecification( );
      // Will remember the concrete type of e, and
      // call the proper version.
   return out;
}
```

Abstract Classes

It is sometimes convenient not to define all virtual functions of the base class.

In that case, it is not possible to declare elements of the base class, because their specification is incomplete.

Such a class is called interface in Java.

Interfaces/ Abstract Classes

An interface 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.

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.
};
```

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.

How does this all work?

Suppose we have a base type T, and derived types $T_1, \ldots, T_n$.

Assume there are two virtual methods in T, for example:

```
virtual int m1( int );
virtual double m2( double );
```

The compiler will define

```
struct vtable_T
{
   int (*m1) int;      // Pointer to function.
   double (*m2) double;
};
```

Function Tables

Every object of type T or T_i has a field

```
vtable_T* rtti;    // Run Time Type Information.
```

Each of the types has one `vtable_T`, which is shared between all objects of the type.

Calling `t. m1( )` means that `t.rtti -> m1( )` is called.

The Run Time Type Information, RTTI) marks to which class the object belongs.

Selecting the proper virtual function costs some small amount of time. This is the reason why the `virtual` keyword exists. It is still more efficient than implementation by hand.

For each type, 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.

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 base classs to a subclass.

It has two posible forms:

```
const T* t = dynamic_cast< T* > ( t );
const T* t = dynamic_cast< const T* > ( t );
   // Result is 0 if the actual type of t does not
   // inherit from T. Can be used for checking.

T& t = dynamic_cast< T& > ( t );
const T& t = dynamic_cast< const T& > ( t );
   // Throws exception bad_cast when actual type of t
   // does not inherit from T. Should be used only
   // when you know that t has proper type.
```

Don't use C-style casts!

Dynamic Cast

The main question about dynamic cast is: When to use it?

Can one do something with dynamic cast, that cannot be done with virtual functions?

1. Don't use `dynamic_cast` to replace the virtual function mechanism! Virtual functions are nicer and more efficient.

2. Sometimes, a method/function is not meaningful on the complete hierarchy, and there is no way to make it meaningful. In that case, one can use `dynamic_cast`. (An example is the rpm field for **washingmaschine**, which really has no meaningful interpretation for other electricdevices.)

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.
}
```

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

If one of the subclasses has a destructor that does something non-trival, then the destructor in the base class must be declared as **virtual**.

Advice: Whenever you want to inherit from some class, declare its destructor as virtual method. You don't know what will be in the hierarchy.

Destructors (2)

```
struct aa { };

struct bb : public aa
{
   std::string s;
   bb( const std::string& s ) : s{s} { }

};

aa* p = new bb{ "my long string" };
delete p;
   // Memory held by the string will not be returned.
```

Adding `virtual ~aa( ) = default;` or `virtual ~aa( ) { }` to the definition of **aa** solves the problem.

Note that naked pointers (as on the last slide) should not be used, because they are not exception safe.

The problem also exists with smart pointers and wrapper classes.

Ownership of Polymorphic Variables

Until now, we only saw cases where the owning variable was not polymorphic:

```
T t;
T1 t1;
T2 t2;
```

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

Ownership of Polymorphic Variables (2)

Polymorphism in local variables is impossible, because the compiler needs to know the size of a stack variable at compile time.

If $T_1, T_2, \ldots, T_n$ inherit from T, then the T_i are possibly bigger than T.

The compiler has no way of knowing the biggest T_i. It cannot even see if there exists something that inherits from T at all.

Polymorphism is also impossible in containers (vector, list, map, unordered_map).

Ownership of Polymorphic Variables (3)

Often one wants to write:

```
T readT( std::istream& );
   // Reads a T1,T2, ...

T t = readT( std::cin );
// Could be T1, T2, ...
```

It is impossible!

Ownership of Polymorphic Variables (4)

Instead one can write:

```
T* readT( std::istream& in )
{
   ... return new T1(  );
   ... return new T2(  );
   ... etc.
}

T* t = readT( std::cin );
```

Possible, but high risk of memory leaks. (Non-normal termination, somebody may later modify the code, etc.)

Ownership of Polymorphic Variables (5)

Two solutions:

1. Write a wrapper class:

```
struct Tholder
{
   T* t;

   Tholder( T* t );
   ~Tholder( ) { delete t; }
};
```

2. Use a smart pointer.

Unique Pointer

unique_pointer can be viewed as a polymorphic wrapper class:

```
template< class X > class unique_ptr
{
    X* p;
    unique_ptr( X& p );
    unique_ptr( unique_ptr<X> && );
    void operator = ( unique_ptr<X> && );
    // There is no copying assignment,
    // and no copy constructor.

    ~unique_ptr( ) { delete p; }
};
```

Write `#include <memory>` at the front of your program.

Unique Pointer (2)

Now one can write:

```
unique_pt<T> readT( std::istream& )
{
   ... return unique_ptr<T> { new T1( ) };
   ... return unique_ptr<T> { new T2( ) };
   ... etc.
}

unique_ptr<T> t = readT( std::cin );
```

There is no risk of memory leak.

Unique Pointer (3)

Unique pointer is nice, but not as flexible as a usual variable.

It is possible to pass a unique pointer to a function, but you have to use `std::move`.

(But you can always pass by reference.)

Unique pointer can be reassigned, but only by using `std::move`.

If you want polymorphic objects with full value semantics, you have to write your own wrapper class.

Defining a Wrapper Class

If one wants more functionality than **unique_ptr** has, one can define a wrapper class:

```
class number
{
   numbase* ref;

   number( const numbase& );
   number( numbase&& );
   number( const number& n );
   number( number&& n );
   void operator = ( const number& n );
   void operator = ( number&& n );
   ~number( );
};
```

Wrapper Class (2)

Make sure that each derived class has the following `clone( ) const` methods:

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

my_real* my_real::clone( ) && override
{
   return new my_real{ std::move( *this ) };
}

...
```

Wrapper Class (3)

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

my_rational* my_rational::clone( ) && const override
{
   return new my_rational{ std::move( *this )};

}

...
```

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

Wrapper Class (4)

The extra `&` in the first version of **clone()** is necessary, because a traditional const method binds both to **const** reference and vvalue reference.

The **override** keyword tells the compiler that the method should have a definition in a base class. You can omit it if you don't like it, but using it gives some protection against typos.

Wrapper Class (5)

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
{
   ...
}
```

(We don't want to distinguish between **const** reference and rvalue reference.)

Wrapper Class (6)

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

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

Class **number**

```
class number
{
   numbase* ref;

   number( const numbase& n )
      : ref{ n. clone( ) } { }

   number( numbase&& n )
      : ref{ std::move(n). clone( ) } { }

   number( const number& n )
      : ref{ n. ref -> clone( ) } { }

   number( number&& n )
      : ref{ std::move( *n.ref ). clone( ) } { }
```

Assignment

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

void operator = ( numbase&& n )
{
   if( &n != ref )
   {  delete ref;
      ref = std::move(n). clone( );
   }
}
```

Assignment (2)

```
void operator = ( number&& n )
   {  std::swap( ref, n. ref ); }

void operator = ( const number& n )
   { *this = number(n); }
```

Destructors

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

Class number must have:

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

R-value Clone

The `clone( ) &&` methods are used when a temporary is copied into the wrapper, for example in the declaration

```
number n{ my_rational{ 1, 2 }};
```

The variable `n` cannot initalize its `ref` with a pointer to the **my_rational**, because it is a temporary, which will be not on on the heap.

In this case, it is useful to have moving **clone( )** methods.

Getting the Contents

In class `number`, one can decide to hide the `number`. 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 number& number::getcontents( ) const { return ref; }
```

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.

Also, think really hard about the question if you really want this.