Templates

Using templates is very easy, we have already seen how to do that:

```
vector< int > aa;
aa. push_back(4);

vector< std::string > s;
s. push_back( "good morning\n" );


std::vector< std::list< int > > x;
x. push_back( std::list< int > :: list( ));
x. back( ). push_back(4);
```

Writing your Own Templates

Writing templates is easy in principle, but there are some details that make it unpleasant:

- Templates have to be written in .h files, because the linker cannot instantiate.
- The compiler can check correctness, only when the template is instantiated.
- Friendship involving templates is hard to define.
- You have to insert `typename` at many places.
- There seems to be no formal definition of the syntax, or it is so complicated that I don't understand it. (more likely) Also, the specification seems to be changing all the time.

Templates: Basic Definition

Let's take `pair` as starting point. A pair of a double and a string can be defined as follows:

```
class pair
{
   double first;
   std::string second;

   pair( const double& first, const std::string& second )
      : first( first ), second( second )
   { }
};
```

Templates

Pairs of other types can be defined similarly, but instead of doing that, it is better to define a generic pair:

```
template < typename X, typename Y >
class pair
{
   X first;
   Y second;

   pair( const X& first, const Y& second )
      : first( first ), second( second )
   { }
};
```

In code, one can declare

`pair<int,int> p; std::pair< std::string, std::string >.`

Templates (2)

The best way to write a template is to first write a concrete class, and then abstract it.

Templates in Bigger Projects

Ideally, one would declare the template in a .h file, and define it in a .cpp file.

The compiler would compile the template, and the linker would combine the code with the code from other files.

Unfortunately, the compiler cannot compile anything because it doesn't know how the template will be instantiated.

As a consequence, the template has to be defined completely in a .h file. Everytime the template is used, the parts that are used will be instantiated and compiled.

Checking Templates

Unfortunately, the compiler can check almost nothing as long as the template is not instantiated.

For example,

```
template< typename X > compare( const X& x1, const X& x2 )
{
   if( x1 < x2 ) ...
      // The compiler cannot check that operator < exists,
      // until the template is instantiated.
}
```

Templates and Friendship

Consider the following file `some_class.h`:

```
#ifndef SOME_CLASS
#define SOME_CLASS 1

class some_class
{
   int something_private;

   friend std::ostream& operator << ( std::ostream&,
                                      const some_class& );
};
```

`operator <<` is friend to that it can print `something_private`.

Templates and Friendship (2)

Now we want to do the same with a template class:

```
#ifndef SOME_CLASS
#define SOME_CLASS

template< typename X >
some_class
{
   X something_private;

   friend std::ostream& operator << ( std::ostream&,
                                      const some_class& c );
};
```

Templates and Friendship (3)

If you write:

```
template< typename X >
std::ostream& operator << ( std::ostream& stream,
                            const some_class< X > & x )
   {  ... stream << x. something_private; }
```

then the result will not compile. The operator will not be `friend` of `some_class< >`.

Templates and Friendship (4)

For non-templates, a `friend` declaration automatically declares the object.

For templates, a `friend` declaration does not declare the function. It must be first declared, and after that made friend.

Now we have a problem: We have to declare `operator <<` before `class some_class`, but the declaration of `operator <<` uses `some_class`.

Templates and Friendship (5)

Solution: Put first an incomplete declaration of `template< > some_class`, then a declaration of `operator << ( )`.

```
#ifndef SOME_CLASS
#define SOME_CLASS 1
template< typename X > class some_class;
   // Incomplete declaration of someclass< >.

template< typename X >
std::ostream& operator << ( std::ostream& stream,
                            const some_class< X > & );
   // Declaration of operator << .

   ... (Rest can be as before)
```

Templates and Friendship (6)

Unfortunately, the code will still not compile, because the `friend` declaration does not make declare friendship to a template `operator <<`, but to a fixed `operator <<`.

It is hard to understand why this is the case. The reason must be the fact that the `friend` declaration is not really a member of the class it occurs in.

If you think about it, there must be also a way to declare non-template functions friends. If friend declarations would be template by default, this would be impossible.

Templates and Friendship (7)

It is sufficient to replace

```
friend std::ostream& operator << (
                            std::ostream& stream,
                            const some_class< X > & );
```

by

```
friend std::ostream& operator << < > (
                            std::ostream& stream,
                            const some_class< X > & );
```

Now the example should compile.

Templates and Friendship (8)

We now have three occurrences of `operator <<`. The types must fit exactly (constness, references), otherwise either the friendship does not work, or the compiler complains about ambiguity between different definitions of `operator <<`.

Typename

Uninstantiated templates contain so little information that the compiler is unable to determine what are types and what are variables:

```
template< typename X >
something( ) const
{
   for( std::vector< X > :: const_iterator
           p =  s. begin( );
           p != s. end( );
           ++ p )
    { ... }
}
```

Typename (2)

The code on the previous slide will not compile. You have to insert `typename` before `std::vector< X > :: const_iterator`. Unfortunately, the compiler nevers tells you that it wants to see `typename`. Instead it gives a kind of syntax error.

Typename (3)

Errors that can be caused by absence of `typename` are:

```
hashtable.h:15: error: expected ; before it
hashtable.h:33: error: expected ‘;’ before p
```

In C^{++}, it is possible to have isolated statements of form `S;` mixed with declarations of form `T v;`. If the compiler does not see that `T` is a type, then it assumes that the statement is of the first form and it expects the `;`.

Smart Pointers

At a couple of places we have seen a similar pattern:

The class object contains a pointer to the heap, and on the heap, the real object is stored.

1. Copying means: Creating a new object on the heap.
2. Assignment: If not self assignment, then clean up the old object on the heap. Create a copy of the value on the heap.
3. Destruction. Clean up the object on the heap.

This pattern is used when the length of the object on the heap is unknown. We used it in vectors, partial objects, and inheritance.

Restoring Object Semantics for Class Hierarchies

We have seen before that run time resolution of virtual functions in C^{++} works only for pointers and for references.

If one doesn' like that, one has to wrap a memory management class around the pointer.

Since the pattern is always the same, why not make it a template?

```
#ifndef MEM_MANAGER
#define MEM_MANAGER 1

template< typename X >
class mem_manager
{
   X* ref;
   mem_manager( const X& );
   mem_manager( const mem_manager&  );
   void operator = ( const X& );
   void operator = ( const mem_manager& );
   ~mem_manager( );
   const X& getcontents( ) const;
   X& getcontents( );
};
```

```
template< typename X >
void mem_manger<X> :: operator = ( const X& x )
{
   delete ref;
   ref = X. clone( );
}
template< typename X >
void mem_manager<X> :: operator = (const mem_manager& m)
{
   if( m. ref != ref )
   {
      delete ref;
      ref = m. ref -> clone( );
   }
}
```

Ownership

It often happens that the same object can be reached in many ways:

```
int x = 4;
p(x);
   // If x is a reference parameter, then inside p we have
   // the reference while variable x still exists.

q( &x );
   // Inside q, have the pointer while variable
   // x still exists.

std::list< int > lll;
std::list< int > :: const_iterator p = lll. begin( );
++ p;
   // *p is reachable through lll and through *p.
```

Ownership (2)

We call the reference will be responsible for cleaning up the object, the owner of the object.

It is not a concept that is mathematically pecise.

If an object can be reached through a direct local variable (which is not a pointer), then the local variable is the owner.

If the object occurs in a container, then the container is the owner.

If an object is referred to by several pointers, then ownership is designers choice.

Smart Pointers: Ownership Strategies (1)

Deep Copy is the strategy that we have been using until now.

There is one pointer to each pointed object. This pointer is also the owner.

Reference Counting. All pointers to the object share ownership. In order to decide when the object gets deleted, one has to count how many pointers point to it. The object gets destroyed when the reference counter goes to 0.

Reference Counter with Copy on Write (COW) All pointers to the object share ownership. When a non-const reference is required, and the reference counter > 1, a new, private copy of the object is made.

Smart Pointers: Ownership Strategies (2)

Reference Linking. Reference linking is like reference counting, but instead of using reference count, one keeps a circular list of references.

Uniqueness. Uniqueness assumes that of every object in the program, there exists at most one copy. This is useful for maintaining identifier spaces, because it makes it possible to check for equality by checking identity.

The memory manager class (that was shown six slides back) can be modified into a each of the smart pointers, dependent on the desired behaviour.

Partiality vs. Totality.

Independent of all of the previous, one has to decide whether one wants totality (the pointer cannot be zero) or partiality. (the pointer can be zero.)

In general, one should not make things too complicated and provide basic functionality only. Some books think that a smart pointer should be able to do everything that a raw pointer can. I don't think so. Designs must be minimal and 'exactly right'.

Design Patterns: Object Factory

An object factory is a method (or a class) that constructs objects whose exact type is known only at run time.

Suppose you have some class hierarchy, and you want to save objects in a file, or read them as input.

```
graphical_object* readfromfile( std::istream& );
graphical_object* readinput( std::istream& );
```

The `clone( ) const` method that we have been using all the time, is an object factory.

Design Patterms: The Visitor

The visitor pattern is a trick for avoiding that related methods get spread through object hierarchies.

This is in principle contrary to the idea of object-oriented programming, but I think it is useful in the following situation:

Usually the low level methods are class dependent, and the higher level methods are more similar again. For those, it makes sense to use a visitor.