Templates

Templates are functions or classes that are parameterized.

We have already seen a few in STL:

```
std::vector< int >
std::vector< double >
std::list< std::vector< >>
std::unordered_map< int, bool >
```

In STL, (as far as I know) the parameters are always types, but this is not necessary. One could also have `vector< T, S >`, a vector of type `T` of size `S`.

Templates are instantiated at compile time.

After instantiation, a template class (or function) is just a usual class (or function), that is compiled in usual fashion.

There is no efficiency price.

Templates are similar to C makros, but better: They have usual scoping rules for names. They use substitution on syntax trees instead of text replacement. They have linker support.

Compiling a Template

1. The compiler reads the definition of the template and stores it.
2. The compiler encounters an instantiation of the template. If everything goes well, the template is instantiated, and compiled in the same way as usual C^{++}. The result is linkable code.
3. The resulting code is linked.

Each of these three steps can create error messages.

Step 1

It would be nice (from theoretical point of view and for the user), if all checking would be done at step 1. Unfortunately, C^{++} does not do that.

The compiler does only syntactic checking, and minimal type checking. I will later explain why. There are good reasons for it.

Step 2

Once the template is instantiated, it has become usual code. It is fully type checked, and compiled in the same way as a direct definition.

Instantiation happens on parse tree level, not on text level.

If you want to write a reliable template class, that doesn't irritate the user (a programmer who uses your template), you have to test it carefully with many different instantiations. Write special instantiation classes, that have exactly the right methods and nothing more.

Templates and Separate Compilation

- A template definition alone cannot be compiled. You would need a fourth type of file for that (in addition to **.o**, **.cpp**, **.h**).
- In order to instantiate a template (e.g. `std::vector<int>`, the compiler must see its definition.

Separate compilation for templates is impossible!

Templates must be defined in **.h** files and included.

Step 3

Because the linker sees a definition of the template in every class where it is used, it has to clean up multiple definitions, (and correct the references). It has to decide which instantiations are equal.

If `std::vector<double>` was used in **file1.cpp** and **file2.cpp**, both files will contain a definition. Only one is needed.

The linker has to detect this, and include only one definition.

General Form

```
template< typename X, typename Y > struct pair
{
   X x;
   Y y;

   pair( )
      { }

   pair( const X& x, const Y& y )
      : x{x}, y{y}
   { }
};
```

Instead of **typename**, one can also use **class**. Use short parameter names that start with a capital.

Using the definition on the previous slide, one can write:

```
pair< int, int > p = { 4,5 };
pair< std::string, int > v{ "good morning", 5 );

p = { 1, 2 }; // OK.
q = { "good evening", 5 }; // OK.
v = { 5,6 };   // Will refuse to compile.
p = { 4, "xxx" }; // Refuses to compile.
```

Error Messages

You have to get used to the error messages. (called template spew or template barf.) They are famous for their length and unreadability.

- Typechecking takes place only when the template definition is fully instantiated. Often, the user (you) don't know the template definition and do not want to see it. The error message makes it seen to you anyway. Often, the template definition is nested. (uses other templates).
- In case a function or method (often `<<`) was not found, the compiler lists all possible candidates. This list is usually long and not helpful.

In my opinion, the main problem is that `g++` should provide error messages in a more structured form, e.g. html. Reversing the order (so that first error comes last) would also help.

Adding Methods

Methods that are defined in the template class definition, are inlined:

```
bool operator == ( const pair& p )
{
   return x == p.x && y == p.y;
}
```

Writing inline methods is the easiest way, but it is not always desirable, because it may make the code too long when the method is called many times. Also, it makes the specification hard to read.

Separating Definition from Declaration

With a usual class, one writes the declarations in the **.h** file, and the definitions in the **.cpp** file. When using templates, both have to be written in the **.h** file.

Write

```
   bool operator == ( const pair& p ) const;
      // In the class definition.
```

Separating Definition from Declaration (2)

Outside of (template) class definition, write:

```
template< typename X, typename Y >
bool pair<X,Y>::operator == ( const pair& p ) const
{
   return x == p.x && y == p.y;
}
```

The linker will not complain about multiple definitions, because it understands that multiple definitions cannot be avoided with templates.

As usual, the method must have a declaration inside the class definition.

Non-member Functions

If you want to define a non-member, template function, this is also possible:

```
template< typename X, typename Y >
bool operator == ( const pair<X,Y> & p1,
                   const pair<X,Y> & p2 )
{
   return p1. x == p2.x && p2. y == p2.y;
}
   // Nicer than using a member function,
   // because it shows the symmetry.
```

You can also write:

```
template< typename X, typename Y >
std::ostream& operator << ( std::ostream& out,
                            const pair< X,Y > & p )
{
   out << "{ " << p.x << ", " << p.y << " }";
   return out;
}
```

Templates and Frienship (1)

Defining methods as friend of template classes is not easy.

Suppose that fields **`x`** and **`y`** of `template< > class pair` are private.

We will have problems defining `operator <<` or `operator ==`, when it is not a member.

Templates and Friendship (2)

The obvious solution does not work:

```
friend bool operator == ( const pair<X,Y> & ,
                          const pair<X,Y> & );

friend std::ostream& operator << ( std::ostream& ,
                                   const pair<X,Y> & );
```

Templates and Friendship (3)

What happens is the following: For non-templates, a **friend** declaration simultaneously declares the function, as if it were declared outside of the class.

Whene `pair<X,Y>` is instantiated, (say with `int`, `std::string`), it declares the instantiated functions and makes them **friend**.

In our example,

```
bool operator == ( const pair<int,std::string> & ,
                   const pair<int,std::string> & );
std::ostream& operator << ( std::ostream& ,
                            const pair<int,std::string> & );
```

In your code, you probably use `==` and `<<` for the same instantiations, the compiler sees the declarations, and accepts your code, but ...

Template and Friendship (3)

there is no definition. Result: Linker errors. Fortunately, g++ gives a useful warning!

Templates and Friendship (4)

We have to make template functions **friend**. It can be done in two ways. First way:

```
template< class X1, class Y1 >
friend std::ostream&
operator << ( std::ostream& , const pair<X1,Y1> & );

template< class X1, class Y1 >
friend bool
operator == ( const pair<X1,Y1> & , const pair<X1,Y1> & );
   // Parameters in friend declaration must be different
   // from parameters of template.
```

Template and Friendship (5)

Second way (works because the functions have the same parameters as the template):

```
   friend std::ostream&
   operator << <> ( std::ostream& , const pair<X,Y> & );

   friend bool
   operator == <> ( const pair<X,Y> & , const pair<X,Y> & );
      // Parameters same as parameters of template.
```

Unfortunately, this is not the end of the story. A **template friend** declaration does not declare the friend. It assumes that the declared function is already declared.

Templates and Friendship (6)

Now we have a circular dependency problem: Functions `template<> operator ==` and `template<> operator <<` must be declared before `template<> class pair`, but use `pair` in their type.

Solution is to use a forward class definition:

```
      template< class X, class Y > struct pair;
```

After that, one can declare:

```
template< class X, class Y >
bool operator == ( const pair<X,Y> & ,
                   const pair<X,Y> & );


template< class X, class Y >
std::ostream& operator << ( std::ostream& ,
                            const pair<X,Y > & );
```

Templates and Friendship (7)

Next comes

```
template< typename X, typename Y >
struct pair
{
   ... Here come the friend declarations
};
```

Did you find this complicated? You are not alone.

Unfortunately, you have to get used to this, if you want to write your own template classes.

All types must fit exactly (constness, references, being template), otherwise either the friendship will not work, the compiler complains about ambiguity between different definitions, or the linker sheds its unreadable tears.

Template Functions

We have already seen template functions in connection to classes, but they can also occur separately:

```
template< typename X > void swap( X& x1, X& x2 )
{
   X y = std::move(x1);
   x1 = std::move(x2);
   x2 = std::move(y);
}
```

Template Functions (2)

You can write

```
int x = 4;
int y = 5;
swap<int>( x, y );
swap( x, y );     // No need to write the types.
                  // Compiler deduces type.
std::string s1 = "morning";
std::string s2 = "good";
swap( s1, s2 );
```

`std::swap` is defined in STL.

Function Template Argument Deduction

Matching: Let $t_1, \ldots, t_n$ be a trees that possibly contain variables at their leaves. Let $u_1, \ldots, u_n$ be trees without variables. A matching Θ of $t_1, \ldots, t_n$ into $u_1, \ldots, u_n$ is a substitution, s.t. $t_i\Theta = u_i$, for each i $(1 \leq i \leq n)$.

Examples

$s(X), t(Y)$ matches into $s(a), t(b)$ with $X = A, Y = b$.

$s(X), t(X)$ does not match into $s(a), t(b)$.

$s(X), t(X)$ matches into $s(a), t(a)$ with $X = A$.

C^{++} uses matching to deduce template arguments.

The types of the function parameters are matched into the types of the calling arguments, where toplevel (not rvalue!) references are first removed:

```
template< typename N1, typename N2, tn N3, tn N4 >
func( N1, std::vector<N1>, const N3*, N4& );

std::list<int> lst;
double n3;
std::string n4;

func( 44, lst, &n3,   n4 );

N1,  std::vector<N1>,  const N3*,  N4   into
int, std::list<int> ,  double* ,   n4
```

```
template< typename X > func( X x );
func(4);   // Matching gives X = int.

template< typename X > func( X& x );
func(4);   // Matching gives X = int, after which
           // function cannot be applied.

template< typename X > swap( X& x1, X& x2 );
{ int i,j;  swap(i,j); }    X = int.
{ int i, double d; swap(i,j); }  Cannot be matched.
{ std::vector<int> v1; std::vector<int> v2; swap(v1,v2)
   X = std::vector<int>.
```

Rvalue References

In case the template argument has **rvalue reference** on top level, the rules are different. Assume that function argument has type `X&&`:

Calling argument	Resulting value of X:
Variable of type Y	$X = Y\&$
Variable of type **const** $Y\&$	$X =$ **const** $Y\&$
Temporary of type Y	$X = Y$
Rvalue reference of type Y	$X = Y$

This is magic that is is essential for perfect forwarding. (Used for example by **emplace**)

Argument Deduction for Constructors

Unfortunately it does not exist.

For example `std::pair( 3.14, 7 )` could easily deduce $X = \textbf{double}, Y = \textbf{int}$. Unfortunately, it won't.

This is the reason why

```
template< typename X, typename Y >
std::pair<X,Y> make_pair( const X& x, const Y& y )
   { return std::pair<X,Y>( x,y ); }
```

exists. It is just a wrapper for the constructor.

There is a proposal to C^{++}-17 to add argument deduction for constructors.

Typename

Consider:

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

It won't compile. The reason for this is that the compiler cannot see if `::field` is a static field, or a type.

It has to be able to construct a syntax tree when the template definition is processed.

Typename (2)

```
template< typename X > f( )
{
   X::bbbb b = X::f;
}

struct aaaa
{
   static int f;
   struct bbbb
   {
      bbbb( int x );
   };
};

f<aaaa> ( );
```

Typename (3)

Insert `typename` before every use of a field of form `t< > ::f` that is a type. (Usually an iterator type.) Unfortunately, the compiler does not always tell you that it wants to see `typename`. Instead it gives a kind of syntax error resulting from the fact that is was not a typename.

Typename (4)

Errors that can be caused by forgetting 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 `;` after `T`.

`class` vs `typename`

`template< typename X >` and `template< class X >` have the same meaning.

I prefer `typename` because `X` can be instantiated with something that is not a class, for example `int`.

It does not matter much. `class` is shorter.

Writing Templates

How to write your own template?

Write an instance class, and give it exactly the methods that you need:

```
struct aaaa
{
};

void operator == ( const aaaa& a1, const aaaa& a2 )
{
}

std::ostream& operator << ( std::cout& out,
                            const aaaa& a );
```

Writing Your Own Templates (2)

Write your template class first for `class aaaa`. After that, introduce the parameter, and test the template for some other instantiations.

Conversion Problems

Consider:

```
template< class X, class Y >
class pair
{
   X x;
   Y y;

public:
   pair( ) { }

   pair( const X& x, const Y& y )
      : x{x}, y{y}
   { }
};
```

Conversion Problems (2)

```
pair<int,int> p = { 4, 5 };
pair< double, double > q = p;
   // Should be possible. (Assuming conversion
   // from int to double is OK.)

if( p == q )
{
   // Should be possible.
   // It is allowed to compare int and double.
};
```

Flexible ==

Solution is easy in principle:

```
template< class X1, class X2, class Y1, class Y2 >
bool operator == ( const pair<X1,Y1> & p1,
                   const pair<X2,Y2> & p2 )
{
   return p1.x == p2.x && p1.y == p2.y;
}
```

If `x,y` are public, there is nothing more to do.

Flexible ==

If you want `operator ==` to be **friend**, you have to proceed as with the simple version of `operator ==`, but only the first method (On slide T&F (4)) works.

Before the definiton of **pair**, write:

```
template< class X, class Y > class pair;


template< class X1, class X2, class Y1, class Y2 >
bool operator == ( const pair<X1,Y1> & ,
                   const pair<X2,Y2> & );
```

Replace **friend** declaration by:

```
template< class X1, class X2, class Y1, class Y2 >
friend bool operator == ( const pair<X1,Y1> & ,
                          const pair<X2,Y2> & );
```

One typo, or forgetting **const** somewhere $\Rightarrow$ pages of error messages!

Converting Assignment

If assignment is not a member, it can be done in the same way as `operator == ( )`.

If it is a member, you can write:

```
template< class X2, class Y2 >
void operator = ( const pair< X2, Y2 > & p )
{
   x = p. x;
   y = p. y;
}
```

Surprising (unpleasant) fact: If fields `x,y` are private, we cannot get `p.x, p.y`.

Making all `pair<X,Y>` Friends

If you want to solve the problem on the previous slide, you have to make all `pair<X,Y>` friend of each other.

In beginning of `class pair`, write

```
template< class X2, class Y2 > friend class pair;
```

Conversion in Construction

Same problem that we had with assignment, also exists with construction/initialization:

```
pair< int, double > p ( 4, 5 );
pair< double, double > q = p;
   // No such constructor.
```

Flexible Construction (2)

Just define one:

```
template< class X2, class Y2 >
pair( const pair< X2, Y2 > & p )
   : x{ p.x }, y{ p.y }
{ }
```

This is a good point to observe the difference between `{ }` and `( )` in initializers.

All instances of `pair` must be friend of each other.

Flexible Construction (3)

Consider:

```
pair< int, int > { 5, 5 };
pair< int, int > q = p;
```

The **pair** is taken apart and reconstructed. If you don't like that, you can write a specialization:

```
pair( const pair& p ) = default;
```

The compiler selects the best fit.

The same can be done with assignment.

Dependent Members

Suppose that we want to define

```
template< class T, class Alloc >
class mylist
{

};
```

There are related classes that are also templates, e.g. **const_iterator**, **iterator**, **node**.

Some of this classes may be public, some may be private. Where to define them?

Dependent Members as Subclasses?

```
template< class T, class Alloc >
class mylist
{
public:
   struct const_iterator {   };
   struct iterator{  };
private:
   struct node{ };
};
```

- Puts public members on the proper place. (user wants to type `mylist<X> :: iterator` .
- Makes it possible to hide private members.
- Most users don't want to see the allocator. Introduces unwanted dependencies, which may lead to problems.

Functions that do not allocate, should not depend on the allocator:

```
template< class T >
void print( std::ostream&,
            mylist< T > :: const_iterator p1,
            mylist< T > :: const_iterator p2 )
{
}

void printdouble( std::ostream&
          mylist<double> :: const_iterator p1,
          mylist<double> :: const_iterator p2 )
{
}
```

Such supporting classes have be defined outside of mylist.

```
template< class T >
struct _Node
{

};    // Hope that user doesn't touch this name.
      // You can still make the constructors private,
      // and make mylist a friend.
template< class T >
struct _Const_iterator
{ };

template< class T >
struct _Iterator
{ };
```

```
template< class T, class Alloc >
class mylist
{

   using const_iterator = _Const_iterator<T> ;
   using iterator = _Iterator<T> ;

   // Now iterators can be exchanged for mylists with
   // same T but different allocators.
};
```

Specialization

Quite often, it happens that a template is too general. For example `template< class B > vector` may have an efficient implementation when B is **bool**. (One could use bitstrings. It is at least space efficient.)

C^{++} allows to write specialized definitions of templates for specialized B.

```
   template< > class vector< bool > {   }
```

You may want to do this, either because of efficiency, or because of a different interface.

Specialization (2)

```
template< typename N > swap(
      std::vector<X> & v1, std::vector<X> & v2 )
{
      v1. swap(v2);
}
```

Note: I am not sure if this works.

Concepts

1. A complete language describing all reasonable constraints on parameter types would be very very complex.
2. It is reasable to use only part of a template class definition. (If a template class has more than one method, you do not have to use all of them). Every reasonable subset has a different set of constraints. You cannot specify them all.

   For example, you can use **pair** without using `operator ==`. This should be possible.

Anyway, there seem to be ongoing discussion about adding a concept language to C^{++}. Such constraints are called concepts. (There is a wikipedia article about them.) I don't think they will be added soon. It seems that $C^{++} - 2017$ is missed.

Concepts (2)

Possible conditions on type X are :

- It must have a default constructor.
- It must be destructable. (This doesn't mean that it has a destructor, but that it is legal to let it go out of scope.)
- It has copy constructor, assignment (moving,non-moving).
- It can be compared.
- It can be printed.

Conclusion

Templates are very powerful, and very nice. If well-used, it is possible to write good quality, very general code with them. But it is not easy. I didn't cover all topics, because I think they are too hard and too specialized. In most cases, I don't know the exact rules, and I solve the problems by trying.

1. You must be willing to type a lot.
2. You must be willing to read large error messages.

If you write a template class, test it with many diverse instantiations.