Templates and Generic Programming

Defining a Template

Function templates

A fucntion template is a formulafrom which we can generate type-specific versions of that function.

- In a templtes definition, the template parameter list cannot be empty
- The compiler uses the deduced templte parameter(s) to **instantiate** a specific version of the function to us

```
template <typename T>
int compare(const T&lhs, const T &rhs)
{
    if (lhs < rhs) return -1;
    if (rhs < lhs) return 1;
    return 0;
}
// instantiates int compare(const int&, const int&);
cout << compare(0, 1) << endl;
```

Nontype Template Parameters

A nontype parameter represent a value rather than a type. When the template is instantiated, nontype parameters are replaced with value supplied by the user or deduced by the complier. These values must be **constant expressions**.

```
template <unsigned N, unsigned M>
int compare(const char (&str1)[N], const char (&str2)[M])
{
    return strcmp(str1, str2);
}

compare("hi", "mom");
// the compiler instantiate
int compare(const char (&)[3], const char (&)[4]);
```

- A nontype parameter may be an integral type, or a pointer or (lvalue) reference to an obj or to a function type. Argument bound to a pointer or reference nontype parameter must have **static lifetime**. A poiner parameter can also be instantiated by a `nullptr` or a zero-valued constant expression
- A template nontype parameter is a constant value inside the templte definition and can be used when constant expressions are required.

- Template programs should try to minimize the number of requirements placed on the argument types.

```
// more type independent and portable by using less

// version of compare that will be correct even if used pointers(comparasion
// between unrelated pointers is UB, but less<T> sort them by phisycal address)
template <typename T>
int compare (const T &lhs, const T&rhs)
{
    if (less<T>()(v1, v2)) return -1;
    if (less<T>()(v2, v1)) return 1;
    return 0;
}
```

Template Compilation

To generate an instantiation, the compiler needs to have the code that defines a function template or class template member function. As a result, defintions of function templates and member functions of class template are ordinarily put into head files.

Templates contains two kinds of names:

- Those that do not depend on a template parameter
- Those that do depend on the template parameter

Class Templates

Class templtes differ from function templates in that the compiler cannot deduce the template parameter type(s) from a class template.

Each instantiation of a class template constitutes an independent class. The type `vector<string>` has no relationship to, or any special access to, the member of any other `vector` type.

If a member function isn't used, it is **not instantiated**. The fact that members are instantiated only if we use them lets us instantiate a class with a type that may not meet the requirements for some of the template's operation.

There is one exception to the rule that we must supply template arguments when we use template type. Inside the scope of the class template itself, we may use the name of the templte without arguments.

```
template <typename T>
class Vec
{
public:
```

```
    Vec operator++(int); // use Vec without argument
};

// the return type appears outside the scope of the class, we must specify the return type u
template <typename T>
Vec<T> Vec<T>::operator++(int)
{
    Vec temp = *this;// we are in the class scope now,
                     //so do not need to repeat the templte argument
    ++*this;
    return tmp;
}
```

Class Templtes and Friends

- a class template that has a nontemplate friend grants that friend access to all the instantiations of the template.
- when the friend itself is a template, the class granting friendship controls whether friendship includes all instantiations of the templte or only specific instantiations(s)

One-to-One Friendship

```
// forward declaration needed for friend dclarations in Blob
template <typename> class BlobPtr;
template <typename> class Blob;
template <typename T>
    bool operator==(const Blob<T>&, const Blob<T>&);

template <typename T> class Blob
{
    friend class BlobPtr<T>;
    friend bool operator==<T>(const Blob<T>&, const Blob<T>&);
};

Blob<char> ca;// BlobPtr<char> and operator==<char> are friends
Blob<int> ia; // BlobPtr<int> and operator==<int> are friends
```

General and Specific Templtate Friendship

```
// forward declaration necessary to befriend a specific instantiation of a template
template <typename T> class Pal;

class C
{
```

```
    friend class Pal<C>; // Pal instantiated with class C is a friend to C

    // all instances of Pal2 are friends
    // no forward declaration required when we befriend all instantiations
    template <typename T>
    friend class Pal2;
};

template <typename T>
class C2
{
    // each instantiation of C2 has the same instance of Pal as a friend
    friend class Pal<T>;

    // all instances of Pal2 are friends of each instance of C2
    template <typename X>
    friend class Pal2;

    // Pal3 is a nontemplate class that is a friend to every instance of C2
    friend class Pal3;
}
```

- To allow all instantiations as friend, the friend declaration must use template parameter(s) differ from those used by the class itself.
- Under the new standard, we can make a template type parameter a friend:

```
template <typename Type> class C{
    friend Type;
}
```

- We can define a type alias for a class template:

```
template<typename T> using twin = pair<T, T>
twin<string> authors;// authors is a pair<string, string>
```

- When we define a template type alias, we can fix one or more of the template parameters:

```
template <typename T> using partNo = pair<T, unsigned>
partNo<string> books; // books is a pair<string, unsigned>
```

Template Parameters

Unlike most other contexts, however, a name used as a template parameter may not be reused with in the templte:

```
typedef double A;
template <typename A, typename B> void f(A a, B b)
```

```
{
    A tmp = a; // tmp has same type as the template parametre A, not double
    double B;  // error: redclares template parameter B
}
```

- Because a parameter name cannot be reused, the name of a templte parameter can appear only once with in a given template parameter list:

```
// error: illegal reuse of template parameter V
template <typename V, typename V> // ...
```

Templte Declarations

As with function parameters, the name of a template parameter need not be the same across the declaration(s) and the definition of the same templte:

```
// declaration
template <typename T> T calc(const T&);
template <typename U> U calc(const U&);

// definition
template <typename Type>
Type calc(const Type &a) { /* ... */ }
```

Declarations for all the templates needed by a given file usually should appear together at the beginning of file before any code that use those names.

Default Template arguments

We can supply **default template arguments** for both function and class templates.

```
template <typename T, typename Compare = less<T>>
int compare (const T &lhs, const &rhs, Compare cmp = Compare())
{
    if (cmp(v1, v2)) return -1;
    if (cmp(v2, v1)) return 1;
    return 0;
}
```

Template Default Arguments and Class Template

Whenever we use a class template, we must always follow the template's name with brackets. In particular, if a class template provides default arguments for all of its template parameters, and we want to use those defaults, we must put an empty bracket pair following the template's name.

```
template <typename T = int> class Numbers{
public:
    Numbers(T v = 0): val(0){}
private:
    T val;
};
Numbers<long double> lots_of_precision;
Numbers<> average_precision;// empty <> says we want the default type
```

Member Templates

A class-either an ordinary class or a class template—may have a member function that itself a template. Such members are refered to as member templates. Member templates may not be virtual

Member Templates of Class Templates

We can also define a member template of a class template. In this case, both the class and the member have their own, independent, template parameters.

```
template <typename Val> class Vec{
    template <typename Iter> Vec(Iter b, Iter e);
};
template <typename Val>
template <typename Iter>
Vec<Val>::Vec(Iter b, Iter e){/* ... */}
```

Unlike ordinary function members of class template, member templates *are* function templates. When we define a member template outside the body of a class template, we must provide the template parameter list for the class templat and the function template. The parameter list for the class template comes first.

Controlling Instantiations

The fact that instantiation are generated when a template is used means that the same instantiation may appear in multiple object files. When two or more separately compiled source files use the same template with the same template arguments, there is an instantiation of that template in each of those files.

- We can avoid this overhead through an **explicit instantiation**. An explicit instantiation has the form

```
extern template declaration;    // instantiation declaration
template declaration;           // instantiation definition
```

For example:

```
extern template class Vec<string>                // declaration
template int compare(const int&, const int&);    // definition
```

- When the compiler seen an `extern` template declaration, it will not generate code for that instantiation in that file.
- Declaring an instantiation as `extern` is promise that there will be a non`extern` use of that instantiation elsewhere in the program.
- There may be several `extern` declarations for a given instantiation but there must be exactly one definition for that instantiation
- Because the compiler automatically instantiates a template when we use it, the `extern` declaration must appear before any code that use the instantiation:

```
extern template class Blob<string>
extern template int compare (const int &, const int &);

Blob<string> sa1, sa2; // instantiation will appear elsewhere

// Blob<int> and its initializer_list ctor instantiated in this file
Blob<int> a1 = {0, 1, 2, 3};
Blob<int> a1 = (a1);// copy ctor instantiated in this file

int i = compare(a1[0], a2[0]);// instantiation will appear elsewhere
```

Instantiation Definition Instantiate All Members

An instantiation definition for a class template instantiates *all* the members of that template including inline members. Consequently, we can use explicit instantiation only for types that can be used with all the members of that template.