Templates and Generic Programming

OOP (通过多态) 和泛型 (Generic)都能处理在编写程序时不知道类型的情况。不同之处在于,

- 动态绑定可以处理类型在运行之前都未知的情况
- 泛型编程中,编译时可以确定类型

Templates are the foundation of generic programming in C++. A template is a blueprint or formula for creating classes or functions. When we use a generic type, such as `vector`, or a generic function, such as `find`, we supply the information needed to transform that blueprint into a specific class or funtion. That transformation happens during compilation.

Defining a Template

Avoid the redundant overloaded function:
`int cmp(const string &a,const string &b)`
`int cmp(const int &a,const int &b)`
`int cmp(const double &a,const double &b)`

> May be similar to `void *` in C to do the same works.
> `int cmp(const void *,const void *)`

Function Templates

A function template is a formula from which we can generate type-specific versions of that function. The template version of `cmp` looks like:

```
template <typename T>
int cmp(const T &a,const T &b){}
```

A template definition starts with the keyword `template` followed by a **template parameter list**, which is a comma-seperated list of one or more **template parameters** bracketed by the `< >`.

The tempalate parameters represent types or values used in the definition of a class or function. When we use a template, we specify-either implicitly or explicitly-**template arguments** to bind to the template parameters.

Our `cmp` function declares one type parameter named `T`. Inside `cmp`, we use the name `T` to refer to a type. Which actual type `T` represents is determined at compiler time based on how `cmp` is used.

Instantiating a Function Template

When we call a function template, the compiler (ordinarily) uses the arguments of the call deduce the template arguments for use. That is, when we call `cmp`, the compiler uses the type of the arguments to determine what type to bind to the template parameter `T`. For example, in this call
`cmp(1,0);// T is int`

The compiler will deduce `int` as the template argument and will bind that argument to the template parameter `T`.

The compiler uses the deduced template parameters to **instantiate** a specific version of the function for us. When the compiler instantiates a template, it creates a new "instance" of the template using the actual template arguments in place of the corresponding template parameters. For example, given the calls

```
cmp(1,0);
vector<int> vec1{1,2,3},vec2{4,5,6};
cmp(vec1,vec2);// T is vector<int>
```

the compiler will instantiate two different version of `cmp`. For the first call, the compiler will **write and compile** a version of `cmp` with `T` replaced by `int`:

```
int cmp(const int &a,const int &b){
    //...
}
```

For the second call, it will generate a version of `cmp` with `T` replaced by `vector<int>`. These compiler-generated functions are generally referred to as an **instantiation** of the template.

We can also do this:

```
    auto f = (int(*)(const int&, const int&))cmp<int>;
    f(3, 5.4);
```

Template Type Parameters

Each type parameter must be preceded by the keyword `class` or `typename`:

```
// error: must preceded U with either typename or class
template<typename T, U> T clac(const T &,const U&);
```

整体来说，`typename`比后者更适用，两者在绝大多数情况都可以互用，仅有少数只可用`typename`。而`class`是先被提出用于模板参数声明的，因此仍有许多程序员延续了`class`指定模板参数。

```
template <typename T>
int cmp(const T &a,const T &b){}

// do the same work
template <class T>
int cmp(const T &a,const T&b){}
```

Nontype Template Parameters

Let's begin with one example.

```
template<size_type N, size_type M>
int cmp(const char (&p1)[N],const char (&p2)[M]){
    return strcmp(p1,p2);
}

cmp("hi","mom");
```

至于nontype template parameters的作用，暂时没有搞清楚。略过。

`inline` and `constexpr` Function Templates

`inline` and `constexpr` sepcify the template functions in the same way as nontemplate functions.

```
// ok, inline specifier follows the template parameter list
template <typename T> inline T min(const T &,const T &);
// error,
inline template <typename T> T min(const T &,const T &);
```

Template Compilation

When the compiler sees the definition of a template, it does not generate code. It generates code only when we instantiate a specific instance of the template. The fact that code is generated only when we uses a template (and not when we define it) affects how we organize our source code and when errors are detected.

Ordinarily, when we call a function, the compiler needs to see only a declaration for the function. Similarly, when we use objects of class type, the class definition must be available, **but the definition of the member functions need not be present**.

Templates are different: To generate an instantiation, the compiler needs to have the code that defines a function template or class template member function.

Class Templates

Class templates differ from function templates in that the compiler cannot deduce the templates parameter types for a class template.

Defining a Class Template

An example:

```
template <typename T> class Blob{
    public:
        typedef T value_type;
```

```
        typedef typename std::vector<T>::size_type size_type;

        Blob();
        Blob(std::initializer_list<T> il);

        size_type size() const{return data->size();}
        bool empty() const {return data->empty();}

        void push_back(const T &t){data->push_back(t);}
        // move version
        void push_back(T &&t){data->push_back(std::move(t));}
        void pop_back();

        // element access
        T &back();
        T &operator[](size_type i);
    private:
        std::shared_ptr<std::vector<T>> data;
        // throws msg if data[i] isn't valid
        void check(size_type i, const std::string &msg)const;
};
```

Instantiating a Class Template

We should add some extra information that is a list of **explicit template arguments** that are bound to the template's parameters. The compiler uses these template arguments to instantiate a specific class from the template.

For example:
`Blob<int> ia;// empty Blob<int>`
`Bloc<int> ia2={0,1,2};// Blob<int> with five elements`

From these definitions, the compiler will instantiate a class that is equivalent to

```
template<> class Blob<int>{
    typedef typename std::vector<int>::size_type size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    //...
    int & operator[](size_type i);
    private:
        std::shared_ptr<std::vector<int>> data;
        void check(size_type i,const std::string &msg) const;
}
```

When the compiler instantiates a class from our `Blob` template, it rewrites the `Blob` template, replacing each instance of the template parameter `T` by the given template argument, which in this case is `int`. **The compiler generates a different class for each element type we specify.

```
// these definitions instantiate two distinct Blob types
Blob<string> names;
Blob<double> prices;
```

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

Member Functions of Class Templates

As with any class, we can define the member funcitons of a class template either inside or outside of the class body. As with any other class, member defined inside the class body are implicitly `inline`.

The definition of a member function outside its class will be like,

```
template<typename T>
ret-type Blob<T>::member-name(param-list)
```

Instantiation of Class-Template Member Funcitons

By default, a member function of a class template is instantiated only if the program uses that member function. For example, this code

```
// instantiates Blob<int> and the initializer_list<int> constructor
Blob<int> squares={0,1,...,9};
// instantiates Blob<int>::size() const
for(size_t i=0;i!=squares.size();++i){
    squares[i]=i*i;// instantiates Blob<int>::operator[](size_t)
}
```

Thus, if a member function isn't used, it is not instantiated. That 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 operations.

Simplifying Use of a Template Class Name inside Class Code

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

```
template <typename T>
class BlobPtr{
    public:
    BlobPtr(){}

    BlobPtr &operator++();
};
```

Here, when we are inside the scope of a class template, the compiler treats references to the template itself as if we had supplied template arguments matching the template's own parameters. That is, it is as if we had written: `BlobPtr<T> &operator++();`

Using a Class Template Name outside the class Template Body

When we define members outside the body of a class template, we must remember that we are not in the scope of the class until the class name is seen:

```
// define members outside the body of a class template
template<typename T>
BlobPtr<T> BlobPtr<T>::operator++(){
    BlobPtr ret=*this;
    ++*this;
    return ret;;
}
```

It is worth noting that `BlobPtr ret=*this` is inside the function body, which means we are in the scope of the class. The compiler assumes that we are using the same type as the member's instantiation.
`BlobPtr<T> ret=*this;`

> Note: Inside the scope of a class template, we may refer to the template without specifying template arguments.

Class Template and Friends

one-to-one frienship

When a class contains a friend, the class and the friend can independently be templates or not. **A class template that has a nontemplate friend grants that friend access to all the instantiations of the template.** When the friend is itself a template,the class granting that friendship controls whether friendship includes all instantiations of the template or only specific instantiations.

An example:

```
// forward declarations needed for friend declarations in Blob
template<typename >
class BlobPtr;

// needed for parameters in operator==
template<typename >
class Blob;

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

    // each instantiation of Blob grants access to the version of
```

```
    // BlobPtr and the equality operator instantiated with the same type
    friend class BlobPtr<T>;
    friend bool operator==<T>(const Blob<T> &,const Blob<T> &);
};
```

More specifically, the friend declarations use `Blob's` template parameter as their own template argument. Thus, **the friendship is restricted to those instantiations of `BlobPtr` and the equality operator that are instantiated with the same type**:

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

The members of `BlobPtr<char>` may access the `nonpublic` parts of `ca` (or any other `Blob<char>` object), but `ca` has no special access to `ia` (or any other `Blob<int>`) or to any other instantiation of `Blob`.

General and Specific Template Friendship

A class can also make every instantiation of another template its friend, or it may limit friendship to a specific instantiation:

```
// forward declaration necessary to be friend a specific instantiation of a template
template<typename T>
class Pal;
// C is an ordinary, nontemplate class
class C{
    // Pal instantiated with class C is a friend to C
    friend class Pal<C>;

    // all instances of Pal2 are friends to C;
    template<typename T> friend class Pal2;
};
template<typename T>
class C2{
    // Pal<T> is friend of C2<T>
    friend class Pal<T>;

    // prior declaration needed
    // all instance of Pal2 are friends of each instance of C2
    // Pal2<X> is friend of C2<T>
    template<typename  X> friend class Pal2;

    // Pal3 is a nontemplate class that is a friend of every instance of C2
    // prior declaration for Pal3 not needed, since Pal3 is a nontemplate class
    friend class Pal3;
};
```

To allow all instantiations as friends, the friend declaration must use template parameters that differ from those used by the class itself, see below:
`template<typname X> frined class Pal2;`

Brfriending the Template's Own Type Parameter

We can make a template type parameter a friend:

```
template<typename T>
class Bar{
    friend T;// grants access to the type used to instantiate Bar
};
```

Thus, for some type named `Foo`, `Foo` would be a friend of `Bar<Foo>`.

It is worth noting that even though a friend ordinarily must be a class or a function, it is okay for `Bar` to be instantiated with a built-in type. Such friendship is allowed so that we can instantiate classes such as `Bar` with built-in type.

Template Type Aliase

`typedef Blob<string> StrBlob;// ok`
`typedef Blob<T> TBlob;// error`

However, we can make a type alias for a class template:

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

`Staic` Members of a Class Templates

Like any other class, a class template can declare `static` members:

```
template<typename T>
class Foo{
    public:
    static std::size_t count()const {return ctr;}
    //...
```

```
    private:
    static std::size_t ctr;
    //...
};
```

Here `Foo` is a class template that has a `public static` member function named `count` and a `private static` data member named `ctr`.

Each instantiation of `Foo` has its own instance of the `static` members. That is, for any given type `X`, there is one `Foo<X>::ctr` and one `Foo<x>::count` member. All objects of type `Foo<X>` share the same `ctr` object and `count` function. E.g.,

```
// instantiate static members Foo<string>::ctr and Foo<string>::count
Foo<string> fs;

// all three objects share the same Foo<int>::ctr and Foo<int>::count members
Foo<int> f1,f2,f3;
```

There must be exactly one definition of each `static` data member of a template class. However, there is a distinct object for each instantiation of a class template. As a result, we define a `static` data member as a template similarly to how we define the member functions of that template:
`template<typename T>`
`size_t Foo<T>::ctr=0;// define and initialize ctr`

When `Foo` is instantiated for a particular template argument type, a **seperate** `ctr` will be instantiated for that class type and initialized to `0`.

To use a `static` member through the class, we must refer to a specific instantiation:

```
Foo<int> fi;// instantiates Foo<int> class
            // and the static data member ctr
auto ct=Foo<int>::count();// instantiates Foo<int>::count
ct=fi.count(); // use Foo<int>::count
ct=Foo::count(); // error, which template instantiation?
```

Like any other member function, a `static` member function is instantiated only if it is used in a program.

Template Parameters

A template papameter name has no intrinsic meaning. We ordinarily name type parameters `T`, but we can use any name:
`template<typename X> X func(...)`
`template<typename FUCK> FUCK func_(...)`

Template Parameters and Scope

Template parameters follow normal scoping rules. However, a name used as a template parameter may not be reused within the template.

```
typedef double A;
template<typename A,typename B>
void f(A a,B b){
    A tmp=a; // tmp has same type as the template parameter A, not double
    double B;// error: redeclares template parameter B
}
```

Normal name hiding says that the `typedef` of `A` is hidden by the type parameter named `A`. Thus, `tmp` is not a `double`. But we cannot reuse names of template parameters, the declaration of the variable named `B` is an error.

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

Template Declarations

A template declaration must include the template parameters:

```
// declares but does not define compare and Blob
template <typename T> int compare(const T&,const T&);
template<typename T> class Blob;
```

As with function parameters, the names of a template parameter need not be the same across the declarations and the definition of the same template:

```
// all three uses of  calc refer to the same function template
template<typename T> T clac(const T &,const T&);// declaration
template<typename U> U calc(const U &,const U&);// declaration
// definition of the template
template<typename Type>
Type clac(const Type &a,const Type &b){/*...*/}
```

Using Class Members That Are Types

Recall that we use the scope operator(`::`) to access both `static` members and type members. In ordinary (nontemplate) code, the compiler has access to the class definition. As a result, it knows that whether a name accessed through the scope operator is a type or a `static` member. For example, when we write `string::size_type`, the compiler has the definition of `string` and can see that `size_type` is a type.

As for template code, what will happen?

When the compiler sees code such as `T::mem` **it won't know until instantiation time** whether `mem` is a type or a `static` data memebr. However, in order to process the template, the compiler must know whether a name represents a type. For example,

`T::size_type *p;`
it needs to know whether we're defining a variable named `p` or are multiplyin g a `static` data member named `size_type` by a variable named `p`.

> By default, the language assumes that a name accessed through the `::` is not a type.

We do so by using the keyword `typename`:

```
template<typename T>
typename T::val_type top(const T &c){
    if(!c.empty())
        return c.back();
    else
        return typename T::val_type();
}
```

Our `top` function expects a container as its argument and uses `typename` to specify its return type and to generate a value initialized element to return if `c` has no elements.

Default Template Arguments

As with the ordinary funciton parameters, we can also supply **default template arguments**.

As an example, we'll rewrite `compare` to use the library `less` function-object template by default:

```
// compare has a default template argument, less<T>
// and a default function argument, F()
template <typename T,typename F=less<T> >
int compare(const T&v1,const T&v2,F f=F()){
    if(f(v1,v2)) return -1;
    if(f(v2,v1)) return 1;
    return 0;
}
```

Here, `F` represents the type of a **callale object** and defined a new function parameter, `f`, that will be bound to a callable object.

The default template argument specifies that `compare` will use the library `less` function-object class, instantiated with the same type parameter as `compare`. The default function argument says that `f` will be a default-initialized object of type `F`.

When uses call this version of `compare`, the may supply their own comparison operation but are not required to do so:

```
bool i=compare(0,42); // use less; i is -1
// result depends on the isbns in item1 and item2
Sales_data item1(cin),item2(cin);
bool j =compare(item1,item2,compareIsbn);
```

```
```

The first call uses the default function argument, which is a default-initialized object of type `less<T>`. In this call, `T` is `int` so that object has type `less<int>`. This instantiation of `compare` will use `less<int>` to do its comparisons.

When `compare` is called with three arguments, the type of the third argument must be a callable object that returns a type that is convertible to `bool` and takes arguments of a type compatible with the types of the first two arguments.

As usual, the types of the template parameters (function template, not class template) are deduced from their corresponding function arguments. In this call, the type `T` is deduced as `Sales_data` and `F` is deduced as the type of `comparaIsbn`.

> 对于一个模板参数，只有当其右侧的所有参数都有默认参数时，它才可以有默认实参。

Template Default Arguments and Class Templates

Q:为什么类模板必须要加上`<>`?
A:

```
vector<int> vec;// 此时才从模板生成代码，生成一个空的vector<int> 对象
vector vec;// 不知道生成什么类型的类对象
```

函数模板是可以由编译器直接推导出模板参数类型，故可写可不写。

Whenever we use a class template, we must always follow the template's name with brackets. **The brackets indicate that a class must be instantiated from a template**. 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:

```
```