



An Idiot's Guide to C++ Templates - Part 2



Ajay Vijayvargiya

1 Jul 2012 CPOL

Let's dwell deeper into C++ templates!

Elevating...

In **first part** of the series, I elaborated following aspects of templates in C++.

- The syntax of C++ templates
- Function Templates and Class Templates
- Templates taking one or more arguments
- Templates taking non-type integral arguments, and taking default arguments.
- The two phase compilation process
- Generic **Pair** and **Array** class that can hold any data-type(s).

In this part, I would try to impart more intriguing concepts of templates, its importance and binding with the other features of C++ language, and would also touch upon STL. No, you need not to know STL at all, and I would not dwell deep into STL. I request you to refresh your *template* understanding by reading first part, before you jump into this one!

- [Requirements from the Underlying Type](#)
 - [Requirements: Function Templates](#)
 - [Requirements: Class Templates](#)
- [Separation of Declaration and Implementation](#)
 - [Separating Class Implementation](#)
- [Templates and Other Aspects of C++](#)
 - [Class Templates, Friends](#)
 - [Class Templates, Operator Overloading](#)
 - [Class Templates, Inheritance](#)
 - [Function Pointers and Callbacks](#)
 - [Templates and Virtual Functions](#)
 - [Templates and Macros](#)
 - [Function Overloading](#)
- [STL - An Introduction](#)
- [Templates and Library Development](#)
 - [Explicit Instantiation](#)
- [Closure](#)

Since my idea remain to elaborate concepts in better and elaborative manner, articles become exhaustive. It takes time and effort, and therefore not everything can be explained in one go. Hence, I write, publish, update - in gradual manner. I request your feedback considering this.

Requirements from the Underlying Type

There are class templates and function templates, and they work on given *type* (template argument type). For example, a function template would sum up two values (or entire array), for type **T** - But this function template would require **operator+** to be present and accessible for the give type (type **T**). Similarly, a class would require the target type to have constructor, assignment operator and other set of required operators.

Requirements: Function Templates

Let me start explaining this topic in little simple and elegant manner. Following function would display value of given *type* on console (using **std::cout**):

```
template<typename T>
void DisplayValue(T tValue)
{
    std::cout << tValue;
}
```

Following set of calls would succeed:

```
DisplayValue(20); // <int>
DisplayValue("This is text"); // <const char*>
DisplayValue(20.4 * 3.14); // <double>
```

Since **ostream** (type of **cout**) has **operator <<** overloaded for all basic types; hence it works for **int**, **char*** and **double** types. There is an *implicit* call to one of those overloads of **operator<<**.

Now, let us define a new structure, having two members in it:

```
struct Currency
{
    int Dollar;
    int Cents;
};
```

And have an attempt to use this type against **DisplayValue** function template:

```
Currency c;
c.Dollar = 10;
c.Cents = 54;

DisplayValue(c);
```

For this call, you will be bombarded with host of errors from your compiler, because the following line fails to compile for the instantiation of **DisplayValue** for type **Currency**:

```
std::cout << tValue; // tValue is now of Currency type
```

Visual C++ will start reporting errors starting with:

```
error C2679: binary '<<': no operator found which takes a right-hand operand of type 'Currency' (or there is no acceptable conversion)
```

GCC compiler would start reporting with:

In function 'void DisplayValue(T) [with T = Currency]':
 16: instantiated from here
 2: error: no match for 'operator<<' in 'std::cout << tValue'

Errors do differ, but they mean the same thing: None of the overloads of overloaded operator **ostream::operator <<** can be called for **Currency**. Both of the compilers do report this simple error in at least **100** lines! The error is not entirely because of **ostream**, nor because of **Currency**, and neither because of templates - but due to assortment of all this into one. At this moment, you have different options available:

- Don't call **DisplayValue** for type **Currency**, and write another function **DisplayCurrencyValue** instead. This is what most programmers would do, after finding their inability to solve the problem with original **DisplayValue** with **Currency** type. Doing this defeats the whole purpose and power of templates in C++. Don't do it!
- Modify **ostream** class, and add new member (i.e. **operator<<**) that takes **Currency** type. But you don't have liberty to do so, since **ostream** is in one of the C++ standard header. However, with global function, which would take **ostream** and **Currency** types, you can do it.
- Modify your own class, **Currency**, so that **cout<<tValue** would succeed.

In short, you need to *facilitate* either of following:

- **ostream::operator<<(Currency value);** (Simplified syntax)
- **ostream::operator<<(std::string value);**
- **ostream::operator<<(double value);**

First version is very much possible, but syntax is slightly complicated. The definition of custom function would take **ostream** as well as **Currency** types, so that **cout<<currency_object;** would work.

Second version demands understanding of **std::string**, and would be slower and complex.

Third version fits the requirement at this moment, and is simplest of other two. It means that **Currency** be converted to **double**, whenever demanded, and the converted data will be passed to **cout** call.

And here is the solution:

```
struct Currency
{
    int Dollar;
    int Cents;

    operator double()
    {
        return Dollar + (double)Cents/100;
    }
};
```

Notice that entire result would come out into a **double** value. So, for **Currency{12, 72}** object, this overloaded operator (function **operator double()**) would return **12.72**. Compiler will now be happy, since there is a **possible conversion** from **Currency** to *one of the types* that **ostream::operator<<** takes.

And therefore, the following call, for type **Currency**:

```
std::cout << tValue;
```

would be expanded as:

```
std::cout << tValue.operator double();
// std::cout << (double)tValue;
```

And hence the following call would work:

```
Currency c;
DisplayValue(c);
```

You see that a simple function call has invoked multiple operations with the help of compiler:

- Instantiating **DisplayValue** for type **Currency**.
- Calling the copy-constructor for class **Currency**, since type **T** is passed by value.
- In an attempt to find best match for **cout<<operator** call, the conversion operator **Currency::operator double** is invoked.

Just add your own copy-constructor code for **Currency** class, and do step-in debugging and see how a simple call is invoking multiple operations!

Alright! Now, let's revisit **PrintTwice** function template from previous part of this article-series:

```
template<typename TYPE>
void PrintTwice(TYPE data)
{
    cout<<"Twice: " << data * 2 << endl;
}
```

The important part is marked in bold. When you call it as:

```
PrintTwice(c); // c is of Currency type
```

The expression **data * 2** in **PrintTwice** will work, since **Currency** type facilitates this possibility. The **cout** statement would be rendered, by the compiler, as:

```
cout<<"Twice: " << data.operator double() * 2 << endl;
```

When you remove the overloaded **operator double()** from class **Currency**, compiler would complain that it doesn't have **operator***, or any *possibility* where this expression can be evaluated. If you change the class (struct) **Currency** as:

```
struct Currency
{
    int Dollar;
    int Cents;

    /*REMOVED : operator double();*/

    double operator*(int nMultiplier)
    {
        return (Dollar+(double)Cents/100) * nMultiplier;
    }
};
```

Compiler will be happy. But doing this will cause **DisplayValue** to fail for **Currency** instantiation. The reason is simple: **cout<<tValue** won't be valid then.

What if you provide **both double** conversion as well as multiplication operator? Would the call to **PrintTwice** fail for **Currency** type? A possible answer would be yes, that the compilation would fail, since compiler would have ambiguity at following the call:

```
cout<<"Twice: " << data * 2 << endl;
// Ambiguity: What to call - conversion operator or operator* ?
```

But no, compilation will **not** fail, and it would make a call to **operator***. Any other possible conversion routines may be called, if compiler doesn't find best candidate. Most of the stuff explained here comes from C++ rulebook, and not explicitly under template umbrella. I explained it for better understanding.

It is generally advised that class shouldn't expose conversion operators too much, instead they should provide relevant functions for them. An example is **string::c_str()**, which returns C-style string pointer; but another implementations (like **CString**) would provide **necessary** conversions implicitly by providing conversion-operators. But for most cases it is recommended that classes should not expose **unnecessary** operators just to make it work with any code. It should provide only reasonable implicit conversions.

Same thing goes with templates and underlying types, the underlying type may provide conversion operator (like conversion to **int** or **string**), but shouldn't provide conversions in excess. Therefore, for **Currency**, only double (and/or to string) conversion would suffice - there is no need to give (binary) **operator*** overloaded.

Moving on. The following function template:

```
template<typename T>
T Add(T n1, T n2)
{
    return n1 + n2;
}
```

would require **T** to have **operator+** which takes argument of same type, and returns same type. In **Currency** class, you would implement it as:

```
Currency operator+(Currency); // No consts, for simplification
```

Interestingly, if you do **not** implement **operator+** in **Currency**, and keep the **double** conversion operator present in class; and call function **Add** as:

```
Currency c1, c2, c3;
c3 = Add(c1,c2);
```

You would get somewhat weird error:

```
error C2440: 'return' : cannot convert from 'double' to 'Currency'
```

Reason is two fold:

- **n1 + n2** is causing **operator double()** be called for both objects passed (implicit conversion). Therefore, **n1+n2** becomes a simple **double+double** expression, resulting into **double** as final value.
- Since the final value is of type **double**, and the same is to be returned from function **Add**; compiler would try to convert this **double** to **Currency**. Since there is no constructor available that takes **double**, hence the error.

For this situation, **Currency** may provide a constructor (a *conversion constructor*) that takes **double**. It makes sense, since **Currency** now provides both TO and FROM **double** conversions. Providing **operator+** just for the sake of **Add** function template doesn't make much of sense.

But, I am not at all against on providing many or all required implicit conversions or overloaded operators. In reality, you **should** provide all required stuff from the underlying type, if the template class/function is designed such way. A class template performing complex number calculation, for example, should be provided with all mathematical operators from the underlying type.

I see most of you are just reading this article, without attempting to do anything. So, here is exercise for you. Find out the type requirements, and fulfill those requirements for **Currency** class for the following function template (taken as-is from first part):

```
template<typename T>
double GetAverage(T tArray[], int nElements)
{
    T tSum = T(); // tSum = 0
    for (int nIndex = 0; nIndex < nElements; ++nIndex)
    {
        tSum += tArray[nIndex];
    }

    // Whatever type of T is, convert to double
    return double(tSum) / nElements;
}
```

Hey, don't be lazy... Come on! Fulfill the requirements of **GetAverage** for **Currency** type. Templates are not theoretical, but very much practical! Don't read ahead, until you understand every bit of text till here.

Requirements: Class Templates

As you should know, class templates would be more prevalent than function templates. There are more class templates in STL, Boost and other standard libraries than function templates. Believe me, there would be more class templates who would craft, than number of function templates you would code.

Class templates would ask for more requirements from the underlying type. Though, it would be mostly dependent on what you demand **from** the class template itself. For example, if you *do not* call **min/max** equivalent method from class- template instantiation, the underlying type *need not* to provide relevant relational operators.

Most class templates would demand following from the underlying type:

- Default constructor
- Copy constructor
- Assignment operator

Optionally, depending on the class template itself (i.e. the purpose of class template), it may also ask for:

- Destructor
- Move constructor and Move assignment operator
(On top of R- Value references)

Only for the sake of simplicity, I would not cover the second group of *basic requirements*, at least not in this article. Let me start off with the first group of basic-requirements understanding from underlying type, followed by explication of more *type* requirements.

Note that all required-methods, when provided by the underlying type, must be accessible (i.e. public) from the class template. For instance, a protected assignment operator from **Currency** class won't help for some collection class that demands it. The assignment of one collection to another collection (both of **Currency**) will not compile due to *protected* nature of **Currency**.

Let's revisit set of classes I used in first part, for this sub-section. Following is a class template **Item**, for holding any type:

```
template<typename T>
class Item
{
    T Data;
public:
    Item() : Data( T() ) {}
    void SetData(T nValue)
    {
        Data = nValue;
    }
    T GetData() const
    {
        return Data;
    }

    void PrintData()
    {
        cout << Data;
    }
};
```

The mandatory requirement, as soon as you instantiate **Item** for particular type, is having default constructor in type **T**. Therefore, when you do:

```
Item<int> IntItem;
```

The type **int** must have default constructor (i.e. **int::int()**), since this class' template constructor would be calling default constructor for underlying type:

```
Item() : Data( T() ) {}
```

We all know that **int** type has its default constructor, which initializes the variable with zero. When you instantiate it for another type, and that type is *not* having default constructor, the compiler would be upset. To understand this, let's craft a new class:

```
class Point
{
    int X, Y;
public:
    // No default constructor
    Point(int x, int y);
};
```

As you know, following call would fail to compile (commented code would compile).

```
Point pt; // ERROR: No default constructor available
// Point pt(12,40);
```

Similarly, following instantiation would also fail:

```
Item<Point> PointItem;
```

When you use **Point** directly, without using non-default constructor, the compiler will report it, and you would be able to correct it - since the source (line) would be reported correctly.

But, when you use it against **Item** class template, the compiler would report the error near the **Item** implementation. For **PointItem** example mentioned above, Visual C++ reports it at following line:

```
Item () : Data( T() )
{}
```

And the error is reported as:

```
error C2512: 'Point::Point' : no appropriate default constructor available
: while compiling class template member function 'Item<T>::Item(void)'
with
[
    T=Point
]
: see reference to class template instantiation 'Item<T>' being compiled
with
[
    T=Point
]
```

Which is very small error message, as far as C++ template error reporting is concerned. Most often, the **first error** would tell the whole story, and the **last error** reference ("*see reference to class...*") shows the actual cause of the error. Depending on your compiler, the error may be easy to understand, or may be quite elusive to get to the actual cause of the error. The quickness and ability of finding and fixing the bug/error would depend on your experience with template programming.

So, to use **Point** class, as a template type parameter for **Item**, you must provide the default constructor in it. Yes, a single constructor taking zero, one or two parameter would also work:

```
Point(int x = 0, int y = 0 );
```

But, the constructor must be public (or accessible, by other means).

Alright. Let's see if **SetData** and **GetData** methods would work for type **Point**. Yes, both of them would work since **Point** has a (compiler provided) copy constructor and assignment operator.

If you implement assignment operator in class **Point** (the underlying type for **Item**), that particular implementation will be called by **Item** (yes, the compiler will generate the relevant code). The reason is simple: **Item::SetData** is assigning the value:

```
void SetData(T nValue) // Point nValue
{
```

```
Data = nValue; // Calling assignment operator.
}
```

If you put the implementation of assignment operator of **Point** class in private/protected area:

```
class Point
{
...
private:
void operator=(const Point&);
};
```

And make a call to **Item::SetData**:

```
Point pt1(12,40), pt2(120,400);
Item<Point> PointItem;

PointItem.SetData(pt1);
```

It would cause a compiler error, starting with something like:

```
error C2248: 'Point::operator =' : cannot access private member declared in class 'Point'
```

```
99:7: error: 'void Point::operator=(const Point&)' is private
```

The compiler would point you the location of error (inside **SetData**), as well as the actual source of error (call from **main**). Actual error messages, and sequence of messages shown would depend on compiler, but most modern compiler will attempt to give detailed error so that you can find the actual source of error.

This example shows that special method of underlying type **T** are called, depending on how/what call is being made. Remember that special members (constructors, assignment operators etc) may be called implicitly - while returning from function, passing value to function, using expressions and so on. It is therefore recommended that underlying type (type **T**) should have these special methods implemented in accessible region of class.

This example also demonstrates how different classes and functions are involved in one function call. In this case, only **Point** class, the class template **Item** and function **main** are involved.

Similarly, when you make a call to **Item<Point>::GetItem** method, the copy constructor would be called (of **Point** class). The reason is simple - **GetItem** is returning a copy of data stored. A call to **GetItem** may not always call copy-constructor, as RVO/NRVO, move-semantics etc may come into picture. But you should always make the copy-constructor of underlying type accessible.

The method **Item<>::PrintData** would **not** succeed for **Point** type, but would succeed for **Currency**, as underlying type for **Item**. Class **Point** doesn't have conversion, or any possible call, to the **cout<<** call. Please do yourself a favor - make **Point** class **cout**-able!

Separation of Declaration and Implementation

Till now I have shown the entire implementation of template-code in one source-file. Treat the *source-file* as one header file, or being implemented in same file containing **main** function. As any C/C++ programmer would know, we put declarations (or say, *interfaces*) in header file, and the respective implementation in one or more source files. The header file would be included by both - respective implementation file, and by one or more client of that *interface*.

At the compilation unit level, the given implementation file would get compiled, and an object file for the same would be generated. When linking (i.e. while generating the final executable/DLL/SO), the linker would gather all those generated object files and would produce final binary image. All good, all fine, unless linker gets upset by missing symbols or duplicate symbols being defined.

Come to templates, it is not exactly the same way. For better understanding let me first throw you some code:

Sample.H

```
template<typename T>
void DisplayValue(T tValue);
```

Sample.CPP

```
template<typename T>
void DisplayValue(T tValue)
{
    std::cout << tValue;
}
```

Main.cpp

```
#include "Sample.H"

int main()
{
    DisplayValue(20);
    DisplayValue(3.14);
}
```

Depending on the compiler and IDE you use, you would put both CPP files for the build. Surprisingly, you will encounter linker errors like:

```
unresolved external symbol "void __cdecl DisplayValue<double>(double)" (??$DisplayValue@N@@YAXN@Z)
unresolved external symbol "void __cdecl DisplayValue<int>(int)" (??$DisplayValue@H@@YAXH@Z)
```

```
(.text+0xee): undefined reference to `void DisplayValue<int>(int)'
(.text+0xfc): undefined reference to `void DisplayValue<double>(double)'
```

When you closely look at the errors, you would find out that linker could not find implementation for following routines:

```
void DisplayValue<int> (int);
void DisplayValue<double> (double);
```

Despite the fact that you have provided implementation of template function **DisplayValue** via source file **Sample.CPP**.

Well, here is the secret. You know that template function gets instantiated only when you make a call to it, with particular data-type(s). The compiler compiles **Sample.CPP** file *separately* having definition of **DisplayValue**. The compilation of **Sample.CPP** is done in separate **translation unit**. Compilation of **Main.cpp** is done in another translation unit. These translation units produce two object files (say **Sample.obj** and **Main.obj**).

When the compiler works on **Sample.CPP**, it does **not find** any references/calls to **DisplayValue**, and it doesn't instantiate **DisplayValue** for any type. Reason is simple, as explained earlier - *On-Demand-Compilation*. Since the translation-unit for **Sample.CPP** doesn't demand any instantiation (for any data-type), it doesn't do second-phase compilation for function template. No object code is generated for **DisplayValue<>**.

In another translation-unit, **Main.CPP** gets compiled and object code gets generated. While compiling this unit, the *compiler* sees a valid interface declaration for **DisplayValue<>**, and performs its job without any issue. Since, we have called **DisplayValue** with two different types, compiler intelligently produces following declarations by itself:

```
void DisplayValue<int>(int tValue);
void DisplayValue<double>(double tValue);
```

And as per its normal behavior, the compiler *assumes* definitions of these symbols in some other translation-unit (i.e. object code), and delegates further responsibility to the linker. This way, Sample.obj and Main.obj files get generated, but none of them do contain implementation of **DisplayValue** - and hence the linker produces set of errors.

What's the solution for this?

The simplest solution, which works for all modern compiler is using the **Inclusion Model**. Another model, not supported by most major compiler vendors is **Separation Model**.

Till now, whenever I explained about template stuff with code written in same file, I used inclusion model. In simple terms, you put all template related code in one file (generally a header file). The client would just include the given header file, and entire code would be compiled in one translation unit. Yet, it would follow on-demand-compilation process.

For the example given above, Sample.H would contain the definition (implementation) of **DisplayValue**:

Sample.H

```
// BOTH: Interface and Implementation
template<typename T>
void DisplayValue(T tValue)
{
    std::cout << tValue;
}
```

Main.CPP would just include this header file. The compiler will be happy, and the linker will also be happy. If you prefer, you may put all declarations, followed by definitions of all functions later in the same file. For example:

```
template<typename T>
void DisplayValue(T tValue);

template<typename T>
void DisplayValue(T tValue)
{
    std::cout << tValue;
}
```

It has following advantages:

- Logical grouping of all declarations, and all implementations.
- No compiler errors if a template function **A** needs to use **B**, and **B** also needs to use **A**. You would have already declared the prototypes for the *other* function.
- Non-inlining of class methods. Till now, I have elaborated entire template class within the class' declaration body. Separating out the method implementation, is discussed later.

Since you can logically divide interface and implementation, you can also *figuratively* divide them into Dot-H and Dot-CPP files:

```
template<typename T>
void DisplayValue(T tValue);

#include "Sample.CPP"
```

Sample.H is giving prototype for **DisplayValue**, and at the end of file, it is including Sample.CPP. Don't worry, it is perfectly valid C++ and would work with your compiler. Note that, your project/build now **must not** add Sample.CPP for compilation process.

The client (Main.CPP) will include the header, which is adding the code of Sample.CPP into it. In this case, just **one** translation unit (for Main.cpp) will do the trick.

Separating Class Implementation

The section demands more attention from the readers. Implementing a method outside a class requires complete type-specification. For example, let's implement **Item::SetData** outside the class definition.

```
template<typename T>
class Item
{
    ...
    void SetData(T data);
};

template<typename T>
void Item<T>::SetData(T data)
{
    Data = data;
}
```

Note the expression **Item<T>** while mentioning the class for which method is being defined. Implementing **SetData** method with **Item::SetData** will not work, since **Item** is not a simple class, but a class template. The symbol **Item** is not a type, but some instantiation of **Item<>** is a type, and therefore the expression **Item<T>**.

For instance, when you instantiate **Item** with type **short**, and use **SetData** method for it,

```
Item<short> si;
si.SetData(20);
```

the compiler would generate *source-code* like this:

```
void Item<short>::SetData(short data)
{
    Data = data;
}
```

Here, the *formed* class name is **Item<short>** and **SetData** is being defined for this class type.

Let's implement other methods outside the class body:

```
template<typename T>
T Item<T>::GetData() const
{
    return Data;
}

template<typename T>
void Item<T>::PrintData()
{
    cout << Data;
}
```

Clearly notice that **template<typename T>** is required in all cases, and **Item<T>** is also required. When implementing **GetData**, the return type is **T** itself (as it should be). In **PrintData** implementation, though **T** is not used, **Item<T>** specification is needed anyway.

And finally, here is the constructor implemented outside class:

```
template<typename T>
Item<T>::Item() /*: Data( T() ) */
{
}
```

Here, the symbol **Item<T>** is the class, and **Item()** is the method of this class (i.e. the constructor). We need not to (or *cannot*, depending on compiler) use **Item<T>::Item<T>()** for the same. Constructor (and destructor) are special member **methods** of class, and not class **types**, and therefore they shouldn't be used as *type*, in this context.

Only for simplicity I commented default initialization of **Data** with the default constructor-call for type **T**. You should uncomment the commented part, and understand the meaning.

If the template class has one or more **default** types/non-types as template parameters, we just need to specify them while declaring the class:

```
template<typename T = int> // Default to int
class Item
{
    ...
    void SetData(T data);
};

void Item<T>::SetData() { }
```

We cannot/need-not specify the default template parameters, at the implementation stage:

```
void Item<T = int>::SetData() {} // ERROR
```

That would simply be an error. The rule and reasoning is very much similar to a C++ function taking default argument(s). We only specify default parameter(s) when declaring the interface of a function, and not when (separately) implementing the function. Eg:

```
void Allocate(int nBytes = 1024);
void Allocate(int nByte /* = 1024* / ) // Error, if uncommented.
{ }
```

Implementing Method Templates Outside Class

For this, first consider a simple example by a code snippet:

```
Item<int> IntItem;
Item<short> ShortItem;
IntItem.SetData(4096);

ShortItem = IntItem;
```

The important line of discussion is the one which is marked bold. It tries to assign **Item<int>** object to **Item<short>** instance, which is not possible, since these two are different types. Of course, we could have used **SetData** on one and **GetData** on another object. But what if we needed the assignment to work?

For this, you can implement a custom assignment operator, which would itself be on top of templates. That would be classified as method template, and it was already [covered in first part](#). This discussion is only for outside-class implementation. Anyway, here is the **in-class** implementation:

```
template<typename U>
void operator = (U other)
{
    Data = other.GetData();
}
```

Where **U** is the type of other class (another instantiation of **Item**). I didn't use **const** and reference specification for **other** argument, only for simplicity. When **ShortItem = IntItem** takes place, following code is generated:

```
void operator = (Item<int> other)
{
    Data = other.GetData();
}
```

Note that **other.GetData()** returns **int**, and not **short**, since source object **other** is of type **Item<int>**. If you call this assignment operator with non-convertible types (such as **int*** to **int**), it would cause compiler error as these two types are not implicitly convertible. One should not use any type of typecasting within template code, for these kind of conversions. Let the compiler report error to the client of your template.

One more interesting thing worth mentioning here. If you code above' assignment operator like this:

```
template<typename U>
void operator = (U other)
{
    Data = other.Data;
}
```

It simply won't compile - the compiler would complain that **Data** is private! You would wonder *why*?

Reason is quite simple: this class (**Item<short>**), and the other class (**Item<int>**) are actually **two different classes** and have no connection between them. By standard C++ rule, only same class can access private data of current class. Since **Item<int>** is another class, it doesn't give private access to **Item<short>** class, and hence the error! That's the reason I had to use **GetData** method instead!

Anyway, here is how we implement a *method template* **outside** the class declaration.

```
template<typename T>
template<typename U>
void Item<T>::operator=( U other )
{
    Data = other.GetData();
}
```

Note that we need to use template specification with **template** keyword two times - one for class template, and one for method template. Following will not work:

```
template<typename T, class U>
void Item<T>::operator=( U other )
```

Reason is simple - class template **Item** **does not** take two template-arguments; it takes only one. Or, if we take it other way around - method template (i.e. assignment operator) doesn't take two template arguments. Class and methods are two separate template-entities, and need to be classified individually.

You should also notice that **<<code>class T>** comes first, followed by **<class U>**. Studying it by the *left-to-right* parsing logic of the C++ language, we see that class **Item** comes first then the method. You may treat the definition as (see tabs):

```
template<typename T>
    template<typename U>
void Item<T>::operator=( U other )
{ }
```

The separation of two-template specification is definitely **not** dependent on the original parameter names used in class and method declarations. By this, I mean **U** and **T** can exchange positions, and it would compile fine. You can also name the way you like - other than **T** or **U**.

```
template<typename U>
template<typename T>
void Item<U>::operator=( T other )
{ }
```

The **order** of arguments must match, however. But, as you can understand, using the same name is recommended for readability.

Read and understand enough? Well, then it is time to test yourself by writing some template code! I just need the following to work:

```
const int Size = 10;
Item<long> Values[Size];

for(int nItem = 0; nItem < Size; ++nItem)
    Values[nItem].SetData(nItem * 40);

Item<float> FloatItem;
FloatItem.SetAverageFrom(Values, Size);
```

The method-template **SetAverageFrom** would calculate the average from the **Item<>** array being passed. Yes, the argument (**Values**) may be of any underlying type **Item**-array. **Implement it outside class body!** Irrespective of who you are - A super-genius in C++ templates or if you think this task as *Rocket-Science* tough, you must do it - Why fool yourself?

Additionally, what would you do if **Values** is an array of underlying type **Currency**?

Most of the template implementations would be using inclusion model only, that too only in one header file, and all inline code! STL, for example, uses header-only, inline implementation technique. Few libraries are using *include-other-stuff* technique - but they require only header to be included by client, and they operate on top of inclusion model only.

For most template related code, *inlining* doesn't harm for few reasons.

One, the template code (class, function, entire-library) is generally short and concise, like implementing a class template **less** which calls **operator <** on the underlying type. Or a collection class which puts and reads the data into collection, without doing too much of laborious work. Most classes would do small and *only-required* tasks like calling a function-pointer/functor, performing string related stuff; and would **not** do intensive calculations, database or file read, sending packet to network, preparing buffer to download and other intensive work.

Two, inlining is just a *request* from the programmer to the compiler and compiler would inline/not-inline the code on its own discretion. It all depends on code complexity, how often it (method) gets called, other possible optimizations around it etc. Linker and profile guided optimization (PGO) also play important role in code optimization, inlining etc. Therefore, putting entire code within class-definition will not do any harm.

Three, not all of the code gets compiled - only the one that gets instantiated would get compiled, and *this* reasoning get more importance because of previous two points mentioned. So, don't worry about code inlining!

When set of class templates along with few *helper* function templates, the code is just like an arithmetic expression for the compiler. For example, you would use **std::count_if** algorithm on a **vector<int>**, passing a functor, which would call some comparison operator. All this, when coupled in single statement, may look complicated and seems processor-intensive. But it is not! The entire expression, even involving different class templates and function templates, is like a simple expression to the compiler - specially in a Release build.

Other model, the **Separation Model**, works on top of **export** keyword, which most compilers don't still support. Neither GCC, nor Visual C++ compiler support this keyword - both compiler would however say this is reserved keyword for future, rather than just throwing non-relevant error.

One concept that logically fits this *modeling* umbrella is **Explicit Instantiation**. I am deferring this concept and I would be elaborating it later. One important thing - Explicit Instantiation and **Explicit Specialization** are two different facets! Both will be discussed later.

Templates and Other Aspects of C++

Gradually, as you would gain firm understanding about the templates, the power of templates in C++ and be passionate about templates, you would get clear picture, that using templates you can craft your own language subset. You can program the C++ language so that it performs some tasks the way you like. You can use and *abuse* the language itself, and ask compiler to generate **source-code** for you!

Fortunately, or unfortunately, this section is not about how to abuse the language and make compiler do *labor*-work for you. This section tells how other concepts like inheritance, polymorphism, operator overloading, RTTI etc. are coupled with templates.

Class Templates, Friends

Any veteran programmer would know the real importance of **friend** keyword. Any newbie or *by-the-books* mortal may detest **friend** keyword, saying that it breaks encapsulation, and the third category would say "*depends*". Whatever your perspective may be, but I believe that **friend** keyword is useful, if judiciously used wherever required. A custom allocator for various classes; a class to maintain the relation between two different classes; or an inner class of a class are good candidates of being **friends**.

Let me first give you an example, where the **friend** keyword along with template is almost indispensable. If you remember the template-based assignment operator in **Item<>** class, you must also recollect that I had to use **GetData** from the *other* object of another type (another variant of **Item<>**). Here is the definition (in-class):

```
template<typename U>
void operator = (U other)
{
    Data = other.GetData();
}
```

The reason is simple: `Item<T>` and `Item<U>` would be different types, where `T` and `U` may be `int` and `short`, for example. One class cannot access private member of another class. If you implement an assignment operator for regular class, you would directly access the data of other object. What would you do to access data of other class (which is, ironically, the same class!) ?

Since the two *specializations* of class template belong to same class, can we make them **friends**? I mean, is it possible to make `Item<T>` and `Item<U>` friends of each other, where `T` and `U` are two different data-types (convertible)?

Logically, it is like:

```
class Item_T
{
    ...
    friend class Item_U;
};
```

So that `Item_U` can access `Data` (private data) of class `Item_T`! Remember that, in reality, `Item_T` and `Item_U` would not be *just* two class-types, but any set of two instantiations on top of class template `Item`.

Self-friendship seems logical, but how to achieve that? Following will simply not work:

```
template<typename t>
class Item
{
    ...
    friend class Item;
};
```

Since `Item` is a class template and not a regular class, therefore symbol `Item` is invalid in this context. GCC reports following:

```
warning: class 'Item<t>' is implicitly friends with itself [enabled by default]
error: 'int Item<int>::Data' is private
```

Amusingly, initially it says *it is implicit friend with itself*, and later it complains about private-access. Visual C++ compiler is more lenient and silently compiles, and makes them friends. Either way, the code is not compatible. We should use code that is portable and indicates `Item` as class template. Since target type is unknown, we cannot replace `T` with any particular data-type.

Following should be used:

```
template <class U>
friend class Item; // No template stuff around 'Item'
```

It forward-declares the class, and implies that `Item` is class template. The compiler is now satisfied, without any warnings. And now, following code works without the penalty to call the function:

```
template<typename U>
void operator = (U other)
{
    Data = other.Data; // other (i.e. Item<U> has made 'me' friend.
}
```

Other than this self-friendship notion, the **friend** keyword would be useful along with templates in many other situations. Of course, it includes regular course of friendships, like connecting a *model* and a *framework* class; or a *manager* class being declared as friend by other *worker* classes. But, in case of templates, an inner class of a template-based outer class may have to make outer class a friend. Another case where template-based base class would be declared as friend by derived class.

At present, I do not have more ready and understandable examples to demonstrate **friend** keyword usage, that are specific to class templates.

Class Templates, Operator Overloading

Class templates would use operator-overloading idea, more often than a regular class would do. A comparator class would use one or more of relational operators, for instance. A collection class would use index-operator to facilitate *get* or *set* operations for element access by index or by key. If you remember class template **Array** from previous part, I used index-operator:

```
template<typename T, int SIZE>
class Array
{
    T Elements[SIZE];
    ...
public:
    T operator[](int nIndex)
    {
        return Elements[nIndex];
    }
};
```

For another example, recollect a class template **Pair** discussed in previous part. So, for example, if I use this class template as:

```
int main()
{
    Pair<int,int> IntPair1, IntPair2;

    IntPair1.first = 10;
    IntPair1.second = 20;
    IntPair2.first = 10;
    IntPair2.second = 40;

    if(IntPair1 > IntPair2)
        cout << "Pair1 is big.";
}
```

That simply won't work and requires **operator >** to be implemented by class template **Pair**:

```
// This is in-class implementation
bool operator > (const Pair<Type1, Type2>& Other) const
{
    return first > Other.first &&
           second > Other.second;
}
```

Though, same thing was already discussed in first part (for **operator ==**), I added this word only for relevance with the concept being elaborated.

Other than these regular overloadable operators, like relational operators, arithmetic operators etc, templates also employ other rarely used overloadable operators: Arrow Operator (**->**) and the Pointer Indirection operator (*****). A simple smart-pointer implementation, on top of class template, illustrates the usability.

```
template<typename Type>
class smart_ptr
{
    Type* ptr;
public:
    smart_ptr(Type* arg_ptr = NULL) : ptr(arg_ptr)
    {}

    ~smart_ptr()
    {
        // if(ptr) // Deleting a null-pointer is safe
        delete ptr;
    }
};
```



```
    }
};
```

The class template `smart_ptr` would hold up a pointer of any type, and would safely delete the memory allocated, in the destructor. Usage example:

```
int main()
{
    int* pHeapMem = new int;
    smart_ptr<int> intptr(pHeapMem);
    // *intptr = 10;
}
```

I have delegated the responsibility of memory *deallocation* to the `smart_ptr` object (`intptr`). When the destructor of `intptr` would get called, it would delete the memory allocated. Note that the first line in main function is just for better clarity. The constructor of `smart_ptr` may be called as:

```
smart_ptr<int> intptr(new int);
```

NOTE: This class (`smart_ptr`) is only for illustration purpose, and it is functionally *not* equivalent to any of standard smart pointer implementations (`auto_ptr`, `shared_ptr`, `weak_ptr` etc.).

Smart pointer would allow any type to be used for safe and sure memory-deallocation. You could also use any UDT:

```
smart_ptr<Currency> cur_ptr(new Currency);
```

After end of current block (i.e. - `{}`), the destructor of `smart_ptr<>` would get called, and would invoke `delete` operator on it. Since the type is known at compile time (instantiation is compile-time!), the destructor of correct type would be invoked. If you put destructor of `Currency`, that would be called as soon as `cur_ptr` ceases to exist.

Coming back on track; how would you facilitate following:

```
smart_ptr<int> intptr(new int);
*intptr = 10;
```

For sure, you would implement pointer indirection (unary) operator:

```
Type& operator*()
{
    return *ptr;
}
```

Distinctly understand that the above definition is non-const implementation, and that is the reason it returns reference of object (`*ptr`, not `ptr`) being held by class instance. Only because of the same, assignment of value `10` is allowed.

Had it been implemented as `const` method, it would not allow assignment to succeed. It would generally return a non-referenced object, or const-reference of the object being held:

```
// const Type& operator*() const
Type operator*() const
{
    return *ptr;
}
```

Following code snippet shows its usage:

```
int main()
{
    smart_ptr<int> intptr(new int);
    *intptr = 10; // Non-const

    show_ptr(intptr);
}
```

```
// Assume it implemented ABOVE main
void show_ptr(const smart_ptr<int>& intptr)
{
    cout << "Value is now:" << *intptr; // Const
}
```

You may like to return `const Type&` for saving few bytes of program stack, from a `const` function. But, in general, class templates do return value types instead. It keeps the design simple, avoids any possible bug from creeping in if underlying type has const/non-const blunder in implementation. It also avoids any unnecessary *reference* creation even from small types (like `int` or `Currency`), which would turn to be more heavier than *value* type returns.

Quite interestingly, you can templatize the `show_ptr` function itself, so that it can display value of any underlying type under the `smart_ptr` object. There is a lot more to explicate about template functions/classes that itself take another template, but needs a separate discussion area for the same. Keeping simple and to-the-point discussion, here is modified `show_ptr`:

```
template<typename T>
void show_ptr(const smart_ptr<T>& ptr)
{
    cout << "Value is now:" << *ptr; // Const
}
```

For `Currency` object, the function will call `Currency::operator double`, so that `cout` will work. Are you awake, or you need to refresh stuff about `cout` and `Currency`? If in confusion, please read that stuff again.

Moving on, lets see what happens when you try to do the following.

```
smart_ptr<Currency> cur_ptr(new Currency);
cur_ptr->Cents = 10;
show_ptr(cur_ptr);
```

The bold line, logically correct, but will fail. Reason is simple - `cur_ptr` is not a pointer, but a normal variable. Arrow operator can only be called if expression on left is a **pointer to structure** (or class). But, as you see, you are using `smart_ptr` as a pointer-wrapper around `Currency` type. Therefore, this should aesthetically work. Essentially, it means, you need to overload arrow operator in class `smart_ptr` !

```
Type* operator->()
{
    return ptr;
}

const Type* operator->() const
{
    return ptr;
}
```

Since I do respect your comfort level with the C++ language, I don't find it necessary to explain about these two different overloads implemented. After the implementation of this operator, `cur_ptr->Cents` assignment will work!

In general, `operator ->` will return a **pointer** only (of some struct/class). But that's not absolutely necessary - `operator ->` may also return reference/value of particular class type. It is not really useful, deep down concept and is rarely implement that way, I don't find it worth discussing.

Do apprehend that overloaded `operator ->` in `smart_ptr` will *not* cause any compile-time error for `smart_ptr<int>`, just because `int` cannot have arrow-operator applied to it. The reason is simple, you will not call this operator on `smart_ptr<int>` object, and hence compiler will not (attempt to) compile `smart_ptr<>::operator->()` for it!

By now, you must have realized importance of operator overloading in C++ and in template arena. Under template domain, there is much more around operators, and it really helps template based development, compiler support, early binding etc.

Class Templates, Inheritance

Before discussing the usability of inheritance along with template-based classes, I would emphasize different modes of inheritance involved. No, it is not about multiple, multilevel, virtual or hybrid inheritance, or base class having virtual functions. The modes are

just around single inheritance:

1. Class template inheriting Regular class
2. Regular class inheriting Class Template
3. Class Template inheriting another Class Template

In template-based class designs, other than single inheritance, **multiple** inheritance would be more frequent than multilevel, hybrid or virtual inheritance. Let me first start off with single inheritance.

You know that *class-template* is a template for a class, which will be instantiated depending on the type(s) and other argument it takes. The instantiation would produce a *template-class*, or more distinctly, **specialization** of that class. The process is known as instantiation, and the outcome is known as specialization.

When inheriting, what would you inherit - a class-template (**Item<T>**), or the specialization (**Item<int>**) ?

These two different models appear same, but are entirely different. Let me give you an example.

```
class ItemExt : public Item<int>
{
}
```

Here you see that normal class **ItemExt** is inheriting from a *specialization* (**Item<int>**), and is not facilitating any other instantiation of **Item**. What does it mean? You might ask.

First consider this : The empty class **ItemExt**, in itself, can be classified as:

```
typedef Item<int> ItemExt;
```

Either way (typedef or inheritance), when you use **ItemExt**, you don't need to (or say, you cannot) specify the type:

```
ItemExt int_item;
```

int_item is nothing but a derived-class object of type **Item<int>**. This means, you cannot create object of other underlying type using the derived class **ItemExt**. The instance of **ItemExt** will always be **Item<int>**, even if you add new methods/members to derived class. The new class may provide other features like printing the value, or comparing with other types etc, but class doesn't allow flexibility of templates. By this, I mean, you cannot do:

```
ItemExt<bool> bool_item;
```

Since **ItemExt** is not a class template, but a regular class.

If you are looking for this kind of inheritance, you can do so - it all depends on your requirements, and design perspective.

Another type of inheritance would be *template-inheritance*, where you would inherit the class template itself and pass the template-parameters to it. Example first:

```
template<typename T>
class SmartItem : public Item<T>
{
};
```

Class **SmartItem** is another class template which is inheriting from **Item** template. You would instantiate **SmartItem<>** with some type, and same type would be passed to class template **Item**. And all this would happen at compile time. If you instantiate **SmartItem** with **char** type, **Item<char>** and **SmartItem<char>** would be instantiated!

As another example of *template-inheritance*, let inherit from class template **Array**:

```
template<size_t SIZE>
class IntArray : public Array<int, SIZE>
{
};
int main()
{
    IntArray<20> Arr;
```

```

    Arr[0] = 10;
}

```

Note that I have used **int** as first template argument, and **SIZE** as second template argument to base class **Array**. The argument **SIZE** is only argument for **IntArray**, and is second argument for base class **Array**. This is allowed, interesting feature and facilitates automatic code generation with the help of compiler. However, **IntArray** would always be array of **ints**, but the programmer may specify the size of array.

Similarly, you may inherit **Array** this way also:

```

template<typename T>
class Array64 : public Array<T, 64>
{
};

int main()
{
    Array64<float> Floats;
    Floats[2] = 98.4f;
}

```

Though, in the examples given above, the derived class itself do not do anything extra, inheritance is very-much required. If you think following template-based **typedef** will do the same, you are wrong!

```

template<typename T>
typedef Array<T, 64> Array64;

typedef<size_t SIZE>
typedef Array<int, SIZE> IntArray;

```

Template based **typedefs** are not allowed. Although I do not see any reason why compilers cannot provide such feature. On top of templates **typedefs** can behave different depending on the context (i.e. based on template parameters). But template-based typedefs at global level are not allowed.

Though, not specific to the template-inheritance discussion, you may achieve typedef without using inheritance also. But in that case too, you need to define a new class.

```

template<size_t SIZE>
struct IntArrayWrapper
{
    typedef Array<int, SIZE> IntArray;
};

```

Usage is slightly different:

```

IntArrayWrapper<40>::IntArray Arr;
Arr[0] = 10;

```

The choice entirely depends on the requirement, flexibility, readability, some coding-standards and by personal choice. The second version is quite cumbersome, in my opinion.

But, if inheritance is desired, and you are going to provide extra features on top of base-template, and/or there is "**is-a**" relationship between base and derived classes, you should use template-inheritance model.

Note that, in almost all cases, template based classes wouldn't have virtual functions; therefore, there is no added penalty on using inheritance. Inheritance is just a data-type modeling, and in simple cases, derived class would also be POD (Plain Old Data). Virtual functions, with templates, will be described later.

By now, I have elaborate two models of inheritance:

- Regular class inheriting class template
- Class template inheriting another class template

I also explicated the difference between template-inheritance (where you pass the template argument(s) to base), and instantiation-inheritance where you inherit from very specific type of template instantiation (called specialization). Note that both **IntArray**

and **Array64** would be classified as template-inheritance, since at least one template-argument is keeping the specialization to happen, and would happen only when derived type is instantiated with specific arguments.

Note that only **ItemExt** is an example of '*Regular class inheriting class template*'. All other examples given are '*class -template inheriting class-template*'.

Now the third type. Can a *class-template inherit a regular class*?

Who said No? Why not!

I don't find or craft any example where base class would be (absolutely) basic, without any stain of template. It is actually unreasonable and would be a bad design to represent "is-a" relationship from a non-template base class. Initially, I thought of giving an example where base class would be singly-linked list, and derived class, based on template would be somewhat smarter (say doubly) linked list.

The bad example:

```
class SinglyLinkedList
{
    // Assume this class implements singly linked list
    // But uses void* mechanism, where sizeof data is
    // specified in constructor.
};

template<class T>
class SmartLinkedList : public SinglyLinkedList
{
};
```

Now, you can say a **SmartLinkedList<>** object **is-a SinglyLinkedList**, which defeats the whole purpose of templates. A template-based class must not depend on non-template class. Templates are abstraction around some data-type for some algorithm, programming-model, a data-structure.

In fact, templates do **avoid** inheritance feature of **OOP**, altogether. It represents most of the abstractions by a single class. By this I do not mean templates would not use inheritance. In fact, many features around templates rely on inheritance feature of C++ - but it would **not** use *inheritance-of-features*, as in classical sense of Object-Oriented-Programming.

Class Templates would use inheritance of rules, inheritance of modeling, inheritance of designs and so on. One example would be make a base class, having copy-constructor and assignment-operator private, without having any data-members in it. Now, you can inherit this *Rule* class, and make all desired classes non-copyable!

Let me finish all major aspects of C++ along with templates, then I would show you some truly intriguing techniques using templates!

Function Pointers and Callbacks

As you do know function-pointer is one of the mechanism in C/C++ language to achieve *dynamic-polymorphism*. Not necessarily, but generally coupled with callback-feature - you set particular user-defined function as a callback, that would be called later. The actual function to be called is determined at runtime, and hence **late-binding** of specific callable function occurs.

To understand, let us consider simple code-snippet.

```
typedef void (*DisplayFuncPtr)(int);

void RenderValues(int nStart, int nEnd, DisplayFuncPtr func)
{
    for(;nStart<=nEnd; ++nStart)
        func(nStart); // Display using the desired display-function
}

void DisplayCout(int nNumber)
{
    cout << nNumber << " ";
}

void DisplayPrintf(int nNumber)
{
}
```

```

    printf("%d ", nNumber);
}

int main()
{
    RenderValues(1,40, DisplayCout); // Address-Of is optional
    RenderValues(1,20, &DisplayPrintf);

    return 0;
}

```

In this code, **DisplayFuncPtr** gives the prototype of the desired function, and is only for better readability. Function **RenderValues** will display the numbers using the given function. I called this function with different callbacks (**DisplayCout** and **DisplayPrintf**) from main function. Late-binding occurs at the following statement.

```
func(nStart);
```

Here **func** may point to either of the two Display-functions (or any other UDF). This type of dynamic-binding has several issues:

- The prototype of callback function must exactly match. If you change **void DisplayCout(int)** to **void DisplayCout(float)**, the compiler will get upset:
*error C2664: 'RenderValues': cannot convert parameter 3 from 'void (__cdecl *)(double)' to 'DisplayFuncPtr'*
- Even though the return value of **func** is not used by **RenderValues**, compiler will not allow any callback function returning non-void.
- And this one troubles me a lot! The **calling convention** must also match. If function specifies **cdecl** as callback function, a function implemented as **stdcall** (**__stdcall**), will not be allowed.

Since function-pointers and callbacks comes from the C language itself, compilers have to impose these restrictions. Compiler just cannot allow incorrect function to avoid call-stack to get corrupted.

And here is template based solution to overcome all the mentioned issues.

```

template<typename TDisplayFunc>
void ShowValues(int nStart, int nEnd, TDisplayFunc func)
{
    for(;nStart<=nEnd; ++nStart)
        func(nStart); // Display using the desired display-function
}

```

You can happily supply any of the functions to **ShowValues** templated-based function:

```

void DisplayWithFloat(float);
int DisplayWithNonVoid(int);
void __stdcall DisplayWithStd(int);

...

ShowValues(1,20, DisplayWithFloat);
ShowValues(1,40, DisplayWithNonVoid);
ShowValues(1,50, DisplayWithStd);

```

Yes, you would get **float** to **int** conversion warning for first function. But return type and calling convention would not matter. In fact, any function that *can be* called with **int** argument would be allowed in this case. You can modify the third function taking double, returning a pointer:

```
int* __stdcall DisplayWithStd(double);
```

The reason is simple. Actual type of **TDisplayFunc** is determined at compile time, depending on the type-of argument passed. In case of function-pointer implementation, there is exactly **one** implementation. But in case of function templates, there would be different instantiations of **ShowValues**, depending on unique function-prototypes you instantiate it with.

Along with the concerns mentioned above for normal C-style function-pointer/callback approach, following are also not allowed as display-function argument:



Functors, i.e. Function objects - A class may implement **operator()** with required signature. For example:

```
struct DisplayHelper
{
    void operator()(int nValue)
    {
    }
};
```

The following code is illegal.

```
DisplayHelper dhFunction;
RenderValues(1,20,dhFunction); // Cannot convert...
```

But when you pass the **dhFunction** (a functor, aka function-object) to function template **ShowValues**, the compiler will make no complains. As I said earlier **TDisplayFunc** may be any type that can be called with **int** argument.

```
ShowValues(1,20, dhFunction);
```



Lambdas - Locally defined functions (C++11 feature). Lambdas will also be not allowed as function-pointer argument to C-style function. Following is erroneous.

```
RenderValues(1,20, [](int nValue)
{
    cout << nValue;
} );
```

But it is perfectly valid for **ShowValues** function template.

```
ShowValues(1,20, [](int nValue)
{
    cout << nValue;
});
```

Of-course, using lambda requires **C++11 compliant** compiler (VC10 and above, GCC 4.5 and above).

Interestingly, the function template may be crafted in different way - where you need not to pass a functor as function parameter. Instead, you can pass it as template argument itself.

```
template<typename TDisplayFunc>
void ShowValuesNew(int nStart, int nEnd)
{
    TDisplayFunc functor; // Create functor here

    for(;nStart<=nEnd; ++nStart)
        functor(nStart);
}

...

ShowValuesNew<DisplayHelper>(1,20); // 1 template, 2 function arguments
```

In this case, I have passed **struct DisplayHelper** as template type argument. The function itself now takes only two arguments. The creation of functor is now done by template function itself. The only disadvantage is that you can now only pass **struct or classes**, having **operator()** defined in it. You cannot pass a normal function to **ShowValuesNew**. You can however pass a lambda's type using **decltype** keyword.

```
auto DispLambda = [](int nValue)
{
    printf("%d ", nValue);
};

ShowValuesNew<decltype(DispLambda)>(1,20);
```

Since the type of any lambda is around **std::function**, which is a class type, and hence object creation (**TDisplayFunc functor;**) is allowed.

By now, you have realized that function-pointer approach is very restrictive. The only advantage is code-size reduction and possibility of putting a function into some library, and later call that function passing different callbacks. The callable callback is truly **late-bound**. Since the core function is defined at one place, compiler does not have much of liberty to optimize the code based on functions (callbacks) passed, especially if core-function resides in other library (DLL/SO). Of course, if the core function is large, and restrictive nature is desired/acceptable, you would use function-pointer approach.

Template based approach, on the other side, do advocate for **early-binding**. Early binding is the core and heart of template based programming. As mentioned before, template code would generally not be intensive and big, like huge data-processing, a gaming engine, batch image processing, security subsystem - but a helper for all these systems. Therefore, early-bound nature actually helps in optimizing the code, since everything is under the compiler-territory.

Templates and Virtual Functions

Virtual functions and templates don't go together - they are into different leagues. Reason is simple - One employs late-binder and other one employs early-binder. Do very well remember that templates are compile-time, unlike **generics** in other managed languages (like C#). The type of a generic is determined at runtime, depending on how it is instantiated. But in case of templates, type is determined at compile time only. There are many other differences, pros and cons of templates and generics, but I am deferring that explanation.

Just for a logical understanding of this separation, consider following code.

```
class Sample
{
public:
    template<class T>
    virtual void Processor() // ERROR!
    {
    }
};
```

It asks **Sample::Processor<T>** method template to be virtual, which does not make any sense. How the sample class is to be used and inherited. So, for example, if you make new class **SampleEx** and inherit it from **Sample**, and attempt implement this virtual function. Which specialization would you override? A **Processor<int>** or **Processor<string>**, for example?

Since there are possibilities of infinite specializations of **Processor** method that can be *overridden*, depending on how method-template **Processor** is being called (via base of any of derived classes) - the **virtual** keyword loses its meaning. Compiler cannot create virtual-function-table for such design. Also, compiler cannot enforce derived class to implement all those infinite implementations, if base class declares the given method-template as **pure-virtual**!

ATL library, from Microsoft, uses template based design on top of inheritance principle - but without virtual functions. For performance reasons, it uses templates, and not virtual functions - it means ATL uses more of static-binding, rather than dynamic-binding.

How would you utilize template based classes, having inheritance, but without virtual-functions? And yet facilitate that base class would know and **call methods of derived** class?

Before I explicate that feature, do remember that such classes will not be complete without derived classes. By this I don't mean abstract classes or pure-virtuals. Class templates, as you know, gets compiled only when instantiated with particular type - and this rule holds true for all of the methods in class. Similar way, base class will not be complete without its partner in crime -derived class. It also means that such classes cannot be exported from a library, whereas normal classes (even abstract) may be exported from a library.

Let me start it with normal inheritance model - a base class, having pure virtual function, and a derived class implementing it.

```
class WorkerCore
{
public:
    void ProcessNumbers(int nStart, int nEnd)
    {
        for (;nStart<=nEnd; ++nStart)
        {
```



```

        ProcessOne(nStart);
    }
}

virtual void ProcessOne(int nNumber) = 0;
};

class ActualWorker : public WorkerCore
{
    void ProcessOne(int nNumber)
    {
        cout << nNumber * nNumber;
    }
};

...

WorkerCore* pWorker =new ActualWorker;
pWorker->ProcessNumbers(1,200);

```

You know that **WorkerCore** class is abstract, and a pointer of this type may point to derived class. **ProcessOne** is the function that would be doing actual work. The binding with actual function (in **ProcessNumbers**) depends where **this** pointer is actually pointing. This is very-much utilizing late-binding feature of the language.

For this trivial task, you don't want heavy runtime penalty - you would prefer early-binding. And there the nifty feature, templates, come to rescue! Carefully understand the following code.

```

template<class TDerived>
class WorkerCoreT
{
public:
    void ProcessNumbers(int nStart, int nEnd)
    {
        for (;nStart<=nEnd; ++nStart)
        {
            TDerived& tDerivedObj = (TDerived&)*this;

            tDerivedObj.ProcessOne(nStart);
        }
    }
};

class ActualWorkerT : public WorkerCoreT<ActualWorkerT>
{
public:
    void ProcessOne(int nNumber)
    {
        cout << nNumber * nNumber;
    }
};

```

First understand the bold ones:

- **TDerived** in base class: Specifies the actual type of derived class. The derived class, when inheriting, must specify it.
- Typcasting in **ProcessNumbers**: Since we only **WorkerCoreT** is actually a **TDerived** object, we can safely typecast **this** to **TDerived**. And then call **ProcessOne** method using the object-reference.
- **<ActualWorkerT>** specification: The derived class itself tells the base that "*Here I am*". This line is important, otherwise type of **TDerived** would be wrong, and so the typecasting.

Important thing to know that **ProcessOne** is not a virtual function, not even a regular member in base class. The base class just *assumes* it exists in derived class, and makes a call to it. If **ProcessOne** doesn't exist in derived class, the compiler will simply raise an error:

- 'ProcessOne' : is not a member of 'ActualWorkerT'

Even though there is typecasting involved, there is no runtime penalty involved, no runtime polymorphism, function-pointer drama etc. The said function exists in derived class, is accessible from base class, and is not restricted to be **void (int)**. It could be, as

mentioned in function-pointers section, **int (float)**, or anything else that can be called with **int** parameter.

The only thing is that a pointer of type **WorkerCoreT** cannot simply point to derived class, and make a successful call to **ProcessOne**. And you can justify that such thing doesn't make sense - either take early binding, or late, not both.

STL - An Introduction

STL stands for Standard Template Library, which is a part of C++ Standard Library. From programmer's point of view, even though it is (optional) *part* of C++ Library, most of other features (classes, functions) are dependent on STL itself. As the "*template*" word suggests, STL is mainly on top of C++ templates - there are class templates and function templates.

STL contains set of **collection classes** for representing arrays, linked lists, trees, sets, maps etc. It also contains **helper functions** to act on container classes (like finding maximum, sum or a particular element), and other auxiliary functions. **Iterators** are important classes that allow iteration over collection classes. First let me give simple example.

```
vector<int> IntVector;
```

Here **vector** is a class template, which is functionally equivalent to arrays. It takes one (mandatory) argument - the *type*. The above statement declares **IntVector** to be a **vector<>** of type **int**. Few points:

- **vector**, along with other elements of STL, comes under **std** namespace.
- To use vector, you need to include **vector** header (and *not* **vector.h**)
- **vector** stores its elements in contiguous memory- meaning that any element can be directly accessed. Yes, very much same as array.

Stepped up example:

```
#include <vector>
int main()
{
    std::vector<int> IntVector;

    IntVector.push_back(44);
    IntVector.push_back(60);
    IntVector.push_back(79);

    cout << "Elements in vector: " << IntVector.size();
}
```

About the bold-marked content:

- The header that must be included to use **vector** class.
- Namespace specification: **std**.
- **vector::push_back** method is used to add elements to **vector**. Initially there are no elements in vector, you insert using **push_back**. Other techniques also exist, but **push_back** is paramount.
- To determine current size (**not** capacity) of vector, we use **vector::size** method. Thus the program will display **3**.

If you were to implement vector, you would implement it like:

```
template<typename Type>
class Vector
{
    Type* pElements; // Allocate dynamically, depending on demands.
    int ElementCount; // Number of elements in vector

public:
    Vector() : pElements(NULL), ElementCount(0)
    {}

    size_t size() const { return ElementCount; };

    void push_back(const Type& element); // Add element, allocate more if required.
};
```

No rocket science here, you know all of it. Implementation of **push_back** would be to allocate additional memory, if required, and set/add the element to given location. This rises obvious question: How much memory to allocate on each new-element insertion? And here comes the **Capacity** subject.

vector also has, not frequently used method: **capacity**. Capacity of **vector** is currently allocated memory (in element count), and can be retrieved using this function. Initial capacity, or the additional memory allocated on each **push_back** depends on implementation (how VC or GCC or other compiler vendors implement it). Method **capacity** will always return *more or equal value*, than the **size** method would return.

I request you to implement **push_back** and **capacity** methods. Add any more data members or methods you may want to add.

One major advantage of vector is that it can be used like standard array; except that the size of array (i.e. element count of vector) is not constant. It may vary. You may think of it as dynamically allocated array, where you allocate desired memory (re-allocate if needed), keep track of size of array, check for memory allocation failure and need to free memory at the end. **std::vector** handles it all, yet for all data-types that meet the "Requirements of this class template".

Having said that vector is functionally equivalent to array, following is valid code. (Yes, there has to be at least 3 elements in vector for this code to work).

```
IntVector[0] = 59; // Modify First element

cout << IntVector[1]; // Display Second element

int *pElement = &IntVector[2]; // Hold Third element
cout << *pElement; // Third
```

It clearly means that **vector** has **operator[]** overloaded, which is like:

```
Type& operator[](size_t nIndex)
{
    return pElements[nIndex];
}

const Type& operator[](size_t nIndex)
{
    return pElements[nIndex];
}
```

I have not shown the basic validations here. Important to note the two overloads based on **const**. Class **std::vector** also has the two overloads - one which returns the **reference** of actual element, other one returns **const-reference**. Former will allow modification of actual element stored(See "Modify First element" comment above), and latter will not allow modification.

What about showing all the elements of a **vector**? Well, the following code will work for **vector<int>** :

```
for (int nIndex = 0 ; nIndex < IntVector.size(); nIndex++)
{
    cout << "Element [" << nIndex <<
        "]" is " << IntVector[nIndex] << "\n";
}
```

Nothing important to explain here, until I explicate the *flaws* with this type of *collection-iteration* code. Anyway, we can utilize function template feature to write-up a function that can display any type of **vector**. Here it is:

```
template<typename VUType> // Vector's Underlying type!
void DisplayVector(const std::vector<VUType>& lcVector)
{
    for (int nIndex = 0 ; nIndex < lcVector.size(); nIndex++)
    {
        cout << "Element [" << nIndex << "]" is " << lcVector[nIndex] << "\n";
    }
}
```

Now this templated vector-iteration code can display any **vector** - **vector<float>**, **vector<string>** or **vector<Currency>**, as long as **cout** can display the type, or the underlying type can make it *cout-able*. Please understand the bold-marked content yourself!

Following code is added only for better grip and understanding.

```
...
IntVector.push_back(44);
IntVector.push_back(60);
IntVector.push_back(79);

DisplayVector(IntVector); // DisplayVector<int>(const vector<int>&);
```

Would the implementation of **DisplayVector** work for all type containers, like sets and maps? It won't! I will cover up it soon.

Another container in STL is **set**. A **set<T>** would store only unique elements of type **T**. You need to include **<set>** header to use it. An example:

```
std::set<int> IntSet;

IntSet.insert(16);
IntSet.insert(32);
IntSet.insert(16);
IntSet.insert(64);

cout << IntSet.size();
```

Usage is very similar to **vector**, except that you need to use **insert** method. Reason is simple and justified: New element may be placed anywhere in **set**, not just at the end - and you can't force an element to be inserted at end.

The output of this code snippet will be **3**, and not **4**. Value **16** is being inserted twice, and **set** will ignore the second insertion request. Only 16, 32 and 64 would exist in **IntSet**.

Well, this article is not about STL, but about templates. I briefed about **set** class also for a reason I am going to explain. You may find relevant documentation, articles, sample code etc on the net for STL. Use following keywords to search your favorites: **vector**, **map**, **set**, **multimap**, **unordred_map**, **count_if**, **make_pair**, **tuple**, **for_each** etc.

Let me bring attention to the subject I have to elaborate.

How would you iterate through all elements of a **set** ? Following code is not going to work for **set**.

```
for (int nIndex = 0; nIndex < IntSet.size(); nIndex)
{
    cout << IntSet[nIndex]; // ERROR!
}
```

Unlike **vector** class, **set** does **not** define **operator[]**. You cannot access any element based on its index - the index doesn't exist for **set**. The order of elements in set are **ascending**: from smaller to larger. There exist weak-strict ordering, comparer class etc, but lets consider his (ascending) as default behavior for the subject in hand.

So, at some point, if elements of **set<int>** are (40,60,80), and later you insert 70, the sequence of elements would become (40, 60, **70**, 80). Therefore, logically, *index* is inappropriate for **set**.

And here comes another important facet of STL: **Iterators**. All container classes do have support for iterators, so that elements of collection can be iterated through. Different kind of iterators are represented by various classes. First let me present you a sample code to iterator a **standard-array**.

```
int IntArray[10] = {1,4,8,9,12,12,55,8,9};

for ( int* pDummyIterator = &IntArray[0]; // BEGIN
      pDummyIterator <= &IntArray[9];    // Till LAST element
      pDummyIterator++)
{
    cout << *pDummyIterator << " ";
}
```

Using simple pointer arithmetic, the code is displaying values of all elements of array. Similarly, *iterators* can be used to iterate a **vector**:

```
vector<int>::iterator lcIter;

for (lcIter = IntVector.begin();
     lcIter != IntVector.end();
     ++lcIter)
{
    cout << (*lcIter);
}
```

Carefully understand about the bold-marked content:

- **iterator** is a class. Specifically a **typedef** inside **vector<int>**. Thus, a variable of type **vector<int>::iterator** may **only** iterate a **vector<int>**, and **not** **vector<float>** or **set<int>**. How exactly iterator is **typedef**'d, shouldn't matter you or any STL programmer.
- **begin** and **end** are methods that return **iterator** of same type. An instance of **vector<Currency>** would return **vector<Currency>::iterator**, when you invoke **begin** or **end** on it.
 - **begin** returns an iterator that points to the first element of container. Think it of as **&IntArray[0]**.
 - Method **end** returns an iterator that points to the **next-to-last** element of container. Think it of as **&IntArray[SIZE]**, where **IntArray** is of **SIZE** size. You know that, for size **10** array, **&IntArray[10]** would be (logically) pointing to next element of **&IntArray[9]**.
 - The expression **++lcIter** calls **operator++** on iterator object, which moves the iterator to point to the next element of collection. It is very much same as **++ptr** pointer arithmetic.
 - The loop starts with **iterator** pointing to **begin**, and goes till it points to **end**.
- The expression ***lcIter** calls the unary **operator*** on iterator, which returns the reference/const-reference of element currently pointed. For example above, it simply returns **int**.

You may not be able to grasp this complex iterator concept, so easily, so soon. You should regularly play with iterators - let the compiler bring you some weird errors, let your program crash down or disturb your debugger and cause assertions. More you bring these errors and assertions, the more you learn!

Exactly the same way, you may iterate a **set**:

```
set<int>::iterator lcIter;
for (lcIter = IntSet.begin();
     lcIter != IntSet.end();
     ++lcIter)
{
    cout << (*lcIter);
}
```

If I ask you to write up iteration-loop for:

- **vector<float>**
- **set<Currency>**
- **vector<Pair>**

Soon you would realize you need to change only **container**-type and/or the **underlying type** and rest of the code remain same! You may tempt to write a function template, which would take the container and the underlying-type as its template type arguments. Something like:

```
template<typename Container, typename Type>
void DisplayCollection(const Container<Type>& lcContainer)
{
    Container<Type>::iterator lcIter;
    for (lcIter = lcContainer.begin(); lcIter != lcContainer.end(); ++lcIter)
    {
        cout << (*lcIter);
    }
}
```

Seems logically correct, but that's *not* going to compile. Similar to **DisplayVector** function, this function attempts to take **lcContainer** argument, having **Container** as collection-class, and its underlying type as **Type**. It won't be easy to understand why it won't work, but it's not that much odd to understand why it won't work.

The syntax of **DisplayVector** was:

```
template<typename VUType>
void DisplayVector(const std::vector<VUType>& lcVector)
```

Where the actual type being passed to function is complete expression: **vector<VUType>&**. The type being passed was not just: **vector&**

The syntax of **DisplayCollection** is something like:

```
template<typename Container, typename Type>
void DisplayCollection(const Container<Type>& lcContainer)
```

Here the type being passed to function (not template) is complete: **Container<Type>&**. Suppose if could call it as:

```
vector<float> FloatVector;

DisplayCollection<vector, float>(FloatVector);
```

The (first) type being passed to *template* is just: **vector**, which is **not** a complete type. Some specialization of **vector** (like **vector<float>**) would make it qualified for being a complete type. Since first template (type) argument cannot be classified as type, we cannot use it that way. Though there exist techniques to pass class-template itself (like just **vector**), and make it complete type based on other arguments/aspects. Anyway, here is modified **DisplayCollection** prototype:

```
template<typename Container>
void DisplayCollection(const Container& lcContainer);
```

Yes, just that simple! But the implementation now demands some changes. So, lets implement it gradually.

```
template<typename Container>
void DisplayCollection(const Container& lcContainer)
{
    cout << "Items in collection: " << lcContainer.size() << "\n";
}
```

All STL containers do have **size** method implemented, and they do return **size_t**. So, irrespective which container is being passed (**set**, **map**, **deque** etc) - the method **size** will work.

The iteration of collection:

```
Container::const_iterator lcIter;

for (lcIter = lcContainer.begin();
     lcIter != lcContainer.end();
     ++lcIter)
{
    cout << (*lcIter);
}
```

Few things to learn:

Since the argument (**lcContainer**) is being passed with **const** qualifier, it is rendered as non-mutable object in this function. It means you cannot insert, delete or (re)assign anything to the container. If a **vector** is being passed, **lcContainer.push_back** would be an error, since object is **const**. Further it means that you cannot iterate it using *mutable iterator*.

- Using **iterator** class, you can change the contents. It is thus referred as mutable-iterator.

- Use **const_iterator** when you don't need to change, or you cannot use mutable-iterator. When object/container itself is const (non-mutable), you must use **const_iterator**.
- Important! An object of **const_iterator** is not same as *constant* object of **iterator**.
That means: **const_iterator != const iterator** - note the space!

How the compiler would return **iterator** or **const_iterator**, when I am calling same methods: **begin** and **end**?

Valid question, and simple answer:

```
class SomeContainerClass
{
    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;
};
```

When object is **const**, the **const** version of method is called - simple C++ rule!

Now, one more important point to consider. The code given above, will not compile on all compilers (specifically the following line):

```
Container::const_iterator lcIter
```

Visual C++ 2008 compiles it fine, but GCC reports following error:

```
error: need 'typename' before 'Container::const_iterator' because 'Container' is a dependent scope
```

To understand the reasoning, consider following class:

```
class TypeNameTest
{
public:
    static int IteratorCount;
    typedef long IteratorCounter;
};

int main()
{
    TypeNameTest::IteratorCounter = 10; // ERROR
    TypeNameTest::IteratorCount var;    //ERROR
}
```

Using **ClassName::Symbol** notation, we can access both **typedef** symbol and statically defined symbol. For simple classes, the compiler is able to distinguish, the programmer is able to resolve and there is no potential ambiguity.

But in case of template functions, where the underlying type itself is dependent on template type argument (like **const_iterator** being based on **Container**), the compiler must be **told** that specified symbol is actually a type, and *not* a static symbol of given class. And there we use **typename** keyword to classify the symbol as a type.

Therefore we should (must) use **typename** keyword:

```
typename Container::const_iterator lcIter; // const_iterator is a type, not static symbol in Container
```

Why can't we use **class** keyword instead, and why does VC++ compile it fine?

Ah! Compiler vendors and their tendency to follow some non-standards. VC++ doesn't need **typename**, GCC needs it (and tells you!). GCC will even accept **class** in-place of **typename**, VC will not accept **class**. But thankfully, both adhere to the standards and do accept **typename** keyword!

The new C++ standard (C++11) brings up some respite, specially while working with STL, templates and complicated iterator definitions. The **auto** keyword. The iteration loop can be modified as:

```
for (auto lcIter = lcContainer.begin(); lcIter != lcContainer.end(); ++lcIter)
{
    cout << (*lcIter);
}
```

The actual type of **lcIter** will be determined automatically, at **compile-time**. You may read about **auto** keyword on the Internet, your favorite author's book or may [refer my article](#).

Templates and Library Development

As you know that the template code doesn't directly go to object file, it gets compiled (second phase compilation) only when instantiated with appropriate template parameters. Since the actual code generation (i.e. **specialization**) happens only by instantiating a function/class template with appropriate template-arguments, the function/class template cannot be exported through a library.

When you attempt to export a function template, such as **DisplayCollection**, through a library (**.LIB**, **.DLL** or **.SO**), the compiler and linker may *depict* that it exported the given function. Linker may throw a warning or error that some symbols (e.g. **DisplayCollection**) was not exported or not found. Since there were no call to function template(s) in the *library itself*, no actual code was generated, and thus nothing actually was exported.

When you later use that library in other project, you would get set of linker errors that some symbols were not found. To recollect this problem, [read this section again](#).

It is therefore **not possible** to export template code from a library, without disclosing the source code, and delivering it (generally via header-files). Though, it is very much possible to expose source-code for only the templated-stuff, and not the core library stuff, which may be playing with void-pointers, and **sizeof** keyword. The core library may actually be made private by exporting it from the library, since core stuff may not be template-based.

External Templates, a feature still pending to be part of C++ standard, is *not* supported by major compilers.

Some libraries may *export* entire class for particular template arguments using **Explicit Instantiation** feature.

Explicit Instantiation

This feature is particularly important if you are exposing your template-based library, either by header-only implementation, or through *wrapper-mode* implementation (hiding core, but exposing features via templates). With *Explicit Instantiation*, you can instruct the compiler (and thus, the linker) to generate the code for specific template arguments. It means you are demanding a *specialization* of template, without actually *instantiating* it in your code. Consider a simple example.

```
template class Pair<int, int>;
```

This statement simply asks the compiler to instantiate **Pair** with **<int,int>** arguments for all method of **Pair** class. That means, compiler will generate code for:

- Data-members - **first** and **second**.
- All three constructors (as mentioned in this and previous article).
- Operators **>** and **==** (as mentioned).

To verify this, you may look at the generated binary (executable, DLL/SO), using appropriate tool. On Windows, you may use Dependency Walker to see if code was generated or not. A simpler method exist to assert if compiler/linker are actually performing explicit instantiation - Let the compiler break on failure. For example:

```
template struct Pair<Currency, int>;
```

Would make the type of **first** as **Currency**. Compiler will attempt to generate code for all methods, and would fail on operator **==**, saying that it (**Currency**) doesn't have operator defined:


```
bool operator == (const Pair<Type1, Type2>& Other) const
{
    return first == Other.first && // ERROR Currency.operator== isn't available.
           second == Other.second;
}
```

It fails on this method, just because it comes before any other method that fails (before `operator<` in this case).

This was just the example to check if compiler is actually generating code for all methods or not. But the main question is: *Why would you want to utilize this feature?*

For example, you expose a string class (like `std::string`, or `CString`). And that class is on top of a template argument - the character type - ANSI or Unicode. A very simple definition of `String` class template:

```
template<typename CharType>
class String
{
    CharType m_str[1024];
public:
    CharType operator[](size_t nIndex)
    {
        return m_str[nIndex];
    }

    size_t GetLength() const
    {
        size_t nIndex = 0;
        while(m_str[nIndex++]);

        return nIndex;
    }
};
```

And a pretty simple usage example:

```
String<char> str;
str.GetLength();
```

And you know that it will produce only following:

- `String<char>::m_str`
- `String<char>::String` - default compiler provided constructor.
- `String<char>::GetLength` method

If you were to put `String` into some library, you may put entire `String` class into header, and ship the header. Here is question is not at all about private-stuff, encapsulation etc, it about **unnecessary** increase in size of different executables produced.

There would be thousands of binaries (DLLs, SOs, executables), and almost all of them would be using `String` class. Wouldn't it be better if you could pack them into one library? Yes, I mean the non-templated *traditional* approach?

To do so, you just ask explicit instantiation for all types you are supposed to export **through library**.

```
template class String<char>;
template class String<wchar_t>;
```

For programmer's convenience, you may typedef different String types. The `std::string` type is, in fact, typedef'd and exported in this manner:

```
typedef basic_string<char, ... > string;
typedef basic_string<wchar_t, ... > wstring;

// Explicit Instantiation
template class /*ATTR*/ basic_string<char, ...>;
template class /*ATTR*/ basic_string<wchar_t, ... >;
```

The base class is **basic_string**, which is class template. Few arguments not shown here only for simplicity, and vendors may have different signature of reset of template arguments (for **basic_string**). Second group shows explicit instantiations for these types. The commented part, `/*ATTR*/` - would depend on compiler vendor. It may be an expression that these instantiations do actually go in library being compiled, or are only acting as header-only. In VC++ implementation, these two instantiations are actually in a DLL.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Ajay Vijayvargiya

Software Developer (Senior)

India 


Started programming with GwBasic back in 1996 (Those lovely days!). Found the hidden talent!

Touched COBOL and Quick Basic for a while.

Finally learned C and C++ entirely on my own, and fell in love with C++, still in love! Began with Turbo C 2.0/3.0, then to VC6 for 4 years! Finally on VC2008/2010.

I enjoy programming, mostly the system programming, but the UI is always on top of MFC! Quite experienced on other environments and platforms, but I prefer Visual C++. Zeal to learn, and to share!

Comments and Discussions

 **19 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/268849/An-Idiots-Guide-to-Cplusplus-Templates-Part-2> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Article Copyright 2012 by Ajay Vijayvargiya
Everything else Copyright © [CodeProject](#), 1999-2020

Web02 2.8.200307.1