An Idiot's Guide to C++ Templates - Part 1

Ajay Vijayvargiya

16 Jan 2013 CPOL

Covering the nitty-gritties of C++ templates.

Prolusion

Most C++ programmers stay away from C++ templates due to their perplexed nature. The excuses against templates:

- Hard to learn and adapt.
- Compiler errors are vague, and very long.
- Not worth the effort.

Admitted that templates are slightly hard to learn, understand, and adapt. Nevertheless, the advantages we gain from using templates would outweigh the negatives. There is a lot more than generic functions or classes that can be wrapped around templates. I would explicate them.

While C++ templates and STL (Standard Template Library) are siblings, technically. In this article, I would only cover templates at the core level. Next parts of this series would cover more advanced and interesting stuff around templates, and some know-how about STL.

Table of Contents

The Syntax Drama

As you probably know, template largely uses the angle brackets: The less than (`<`) and the greater than (`>`) operators. For templates, they are always used together in this form:

```
< Content >
```

Where `Content` can be:

1. `class T` / `typename T`
2. A data type, which maps to `T`
3. An integral specification
4. An integral constant/pointer/reference which maps to specification mentioned above.

For point 1 and 2, the symbol `T` is nothing but some data-type, which can be any data-type - a basic datatype (`int`, `double` etc), or a UDT.

Let's jump to an example. Suppose you write a function that prints double (twice) of a number:

```
void PrintTwice(int data)
{
    cout << "Twice is: " << data * 2 << endl;
}
```

Which can be called passing an `int`:

```
PrintTwice(120); // 240
```

Now, if you want to print double of a `double`, you would overload this function as:

```
void PrintTwice(double data)
{
    cout << "Twice is: " << data * 2 << endl;
}
```

Interestingly, class type `ostream` (the *type* of `cout` object) has multiple overloads for `operator <<` - for all basic data-types. Therefore, same/similar code works for both `int` and `double`, and no change is required for our `PrintTwice` overloads - yes, we just *copy-pasted* it. Had we used one of `printf`-functions, the two overloads would look like:

```
void PrintTwice(int data)
{
    printf("Twice is: %d", data * 2 );
}

void PrintTwice(double data)
{
    printf("Twice is: %lf", data * 2 );
}
```

Here the point is not about `cout` or `print` to display on console, but about the code - which is **absolutely same**. This is *one* of the many situations where we can utilize the groovy feature provided by the C++ language: Templates!

Templates are of two types:

- **Function Templates**
- **Class Templates**

C++ templates is a programming model that allows *plugging-in* of any data-type to the code (templated code). Without template, you would need to replicate same code all over again and again, for all required data-types. And obviously, as said before, it requires code maintenance.

Anyway, here is the *simplified* `PrintTwice`, utilizing templates:

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

Here, actual *type* of `TYPE` would be deduced (determined) by the compiler depending on argument passed to the function. If `PrintTwice` is called as `PrintTwice(144);` it would be an `int`, if you pass `3.14` to this function, `TYPE` would be deduced as `double` type.

You might be confused what `TYPE` is, how the compiler is going to determine that this is a function template. Is `TYPE` type defined using `typedef` keyword somewhere?

No, my boy! Here we use the keyword **`template`** to let compiler know that we are defining a function template.

Function Templates

Here is the *templated* function `PrintTwice`:

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

The first line of code:

`template<class TYPE>`

tells the compiler that this is a *function-template*. The actual meaning of `TYPE` would be deduced by compiler depending on the argument passed to this function. Here, the name, `TYPE` is known as **template type parameter**.

For instance, if we call the function as:

```
PrintTwice(124);
```

`TYPE` would be replaced by compiler as `int`, and compiler would **instantiate** this template-function as:

```
void PrintTwice(int data)
{
    cout<<"Twice: " << data * 2 << endl;
}
```

And, if we call this function as:

```
PrintTwice(4.5547);
```

It would instantiate another function as:

```
void PrintTwice(double data)
{
    cout<<"Twice: " << data * 2 << endl;
}
```

It means, in your program, if you call `PrintTwice` function with `int` and `double` parameter types, **two** instances of this function would be generated by compiler:

```
void PrintTwice(int data) { ... }
void PrintTwice(double data) { ... }
```

Yes, the code is duplicated. But these two overloads are instantiated by the compiler and not by the programmer. The true benefit is that you need not to do *copy-pasting* the same code, or to manually maintain the code for different data-types, or to write up a new overload for new data-type that arrives later. You would just provide a **template** of a function, and rest would be managed by compiler.

It is also true that code size would increase, since there are now two function definitions. The code-size (at binary/assembly level) would almost be same. Effectively, for **N** number of data-types, **N** instances of same function (i.e. overloaded functions) would be created. There are advanced compiler/linker level optimizations which can somewhat reduce the code size, if instantiated functions are same, or some part of function body is same. I wouldn't discuss it now.

But, on a positive side, when you manually define **N** different overloads (say `N=10`), those **N** different overloads would be anyway be compiled,linked and packed in binary (the executable). However, with templates, *only* the required instantiations of function would get into final executable. With templates, the overloaded copies of function might be less than N, and it can be more than N - but exactly the number of required copies - no more no less!

Also, for non-templated implementations, the compiler has to compile all those N copies - since they are in your source-code! When you attach *template* with a generic function, compiler would compile only for required set of data-types. It basically means the compilation would be faster if number of different data-types is less than **N**!

It would be a perfectly valid argument, that the compiler/linker would likely do all possible optimizations to remove unused non-template function' implementations from the final image. But, again, do understand that compiler has to **compile** all those overloads (for syntax checking etc). With templates, the compilation would happen only for required data-types - you can call it as "*On demand compilation*".

Enough of text-only content for now! You can come back and re-read it again. Let's move ahead.

Now, let's write another function template that would return the twice of given number:

```
template<typename TYPE>
TYPE Twice(TYPE data)
{
   return data * 2;
}
```

You should have noticed that I used **`typename`**, instead of `class`. No, it is not required to use `typename` keyword if a function returning something. For template programming, these two keywords are very much the same. There is a historical reason for having two keywords for same purpose, and I hate history.

However, there are instances where you can *only* use the newer keyword - **`typename`**. (When a particular type is defined in another type, and is dependent on some template parameter - Let this discussion be deferred to another part).

Moving ahead. When we call this function as:

```
cout << Twice(10);
cout << Twice(3.14);
cout << Twice( Twice(55) );
```

Following set of functions would be generated:

```
int     Twice(int data) {..}
double  Twice(double data) {..}
```

Two things:

- In third line of code snipped above, `Twice` is called *twice* - the return value/type of first call would be the argument/type of second call. Hence, both calls are of **`int`** type (Since argument type `TYPE`, and return type are same).
- If a template function is instantiated for a particular data-type, compiler would re-use the same function' instance - if the function is invoked again for same data-type. It means, irrespective of where, in your code, you invoke the function template with same type - in same function, in different function, or anywhere in another source file (of same project/build).

Let's write up a function template that would return the addition of two numbers:

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

Firstly, I just replaced template-type parameter's name - `TYPE` with symbol **`T`**. In template programming, you would generally use `T` - but that's a personal choice. You should better use a name that reflects the meaning of type-parameter, and that improves code readability. This symbol can be any name which follows variable naming rules in the C++ language.

Secondly, I re-used the template parameter `T` - for both of arguments (`n1` and `n2`).

Let's slightly modify `Add` function, which would store the addition in local variable and then return the calculated value.

```
template<class T>
T Add(T n1, T n2)
{
    T result;
    result = n1 + n2;

    return result;
}
```

Quite explanatory, I used the type parameter **T** within the function's body. You might ask (you should): "*How the compiler would know what is type of* `result`, *when it tries to compile/parse the function* `Add`*?*"

Well, while looking at the body of function template (`Add`), compiler would **not** see if `T` (template type parameter) is correct or not. It would simply check for basic syntax (such as semi-colons, proper usage of keywords, matching braces etc), and would report errors for those basic checks. Again, it depends on compiler to compiler how it handles the template code - but it would not report any errors resulting due to template type parameters.

Just for completeness, I would reiterate, compiler will not check if (currently relevant only for function `Add`):

- `T` is having a default constructor (so that `T result;` is valid)
- `T` supports the usage of `operator +` (so that `<code>n1+n2` is valid)
- `T` has an *accessible* copy/move-constructor (so that `return` statement succeeds)

Essentially, the compiler would have to compile the template code in two phases: Once for basic syntax checks; and later for **each instantiation** of function template - where it would perform actual code compilation against the template data-types.

It is perfectly okay if you did not completely understood this two phase compilation process. You would get firm understanding as you read through this tutorial, and then you would come back to read these theory-sessions later!

Pointers, References and Arrays with Templates

First a code sample (No worries - it is simple code snippet!):

```
template<class 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;
}


int main()
{
    int  IntArray[5] = {100, 200, 400, 500, 1000};
    float FloatArray[3] = { 1.55f, 5.44f, 12.36f};

    cout << GetAverage(IntArray, 5);
    cout << GetAverage(FloatArray, 3);
}
```

For the first call of `GetAverage`, where `IntArray` is passed, compiler would instantiate this function as:

```
double GetAverage(int tArray[], int nElements);
```

And similarly for `float`. The return type is kept as `double` since average of numbers would logically fit in `double` data-type. Note that this is just for this example - the actual data-type that comes under `T` may be a class, which may not be converted to `double`.

You should notice that a function template may have template type arguments, along with non-template type arguments. It it not required to have all arguments of a function template to arrive from template types. `int nElements` is such function argument.

Clearly note and understand that template type-parameter is just **T**, and not **T*** or `T[]` - compilers are smart enough to deduce the type `int` from an `int[]` (or `int*`). In the example given above, I have used `T tArray[]` as an argument to function template, and actual data-type of `T` would intelligently be determined from this.

Most often, you would come across and would also require to use initialization like:

```
T tSum = T();
```

First thing first, this is not template specific code - this comes under C++ language itself. It essentially means: Call the **default constructor** for this datatype. For `int`, it would be:

```
int tSum = int();
```

Which effectively initializes the variable with `0`. Similarly, for `float`, it would set this variable to `0.0f`. Though not covered yet, if a user defined class type comes from `T`, it would call the default constructor of that class (If callable, otherwise relevant error). As you can understand that `T` might be any data-type, we cannot initialize `tSum` simply with an integer zero (`0`). In real case, it may be a some string class, which initializes it with empty string (`""`).

Since the template type `T` may be any type, it must also have **`+= operator`** available. As we know, it is available for all basic data types (`int`, `float`, `char` etc.). If actual type (for `T`), doesn't have **+=** operator available (or any possibility), compiler would raise an error that actual type doesn't have this operator, or any possible conversion.

Similarly, the type `T` must be able to convert itself to `double` (see the `return` statement). I will cover up these nitty-gritties, later. Just for better understanding, I am re-listing the required *support* from type **T** (now, only applicable for `GetAverage` function template):

- Must have an *accessible* default constructor.
- Must have `+= operator` callable.
- Must be able to convert itself to `double` (or equivalent).

For `GetAverage` function template prototype, you may use `T*` instead of `T[]`, and would mean the same:

```
template<class T>
GetAverage(T* tArray, int nElements){}
```

Since the caller would be passing an array (allocated on stack or heap), or an address of a variable of type `T`. But, as you should be aware, these rules comes under rule-book of C++, and not specifically from template programming!

Moving ahead. Let's ask the *actor* '*reference*' to come into template programming *flick*. Quite self-explanatory now, you just use **T&** as a function template argument for the underlying type `T`:

```
template<class T>
void TwiceIt(T& tData)
{
    tData *= 2;
    // tData = tData + tData;
}
```

Which calculates the twice value of argument, and puts into same argument' value. You would call it simply as:

```
int x = 40;
TwiceIt(x); // Result comes as 80
```

Note that I used `operator *=` to get twice of argument `tData`. You may also use `operator +` to gain the same effect. For basic data-types, both operators are available. For class type, not both operators would be available, and you might ask the class' to implement required operator.

In my opinion, it is logical to ask `operator +` be defined by class. The reason is simple - doing `T+T` is more appropriate for most UDTs (User Defined Type), than having `*= operator` . Ask yourself: What does it mean if some class `String` or `Date`

implements, or is asked, to implement following operator:

```
void operator *= (int); // void return type is for simplicity only.
```

At this point, you now clearly understand that template parameter type `T` may be inferred from `T&`, `T*` or `T[]`. Therefore, it is also possible and very *reasonable* to add **`const`** attribute to the parameter which is arriving to function template, and that parameter would not be changed by function template. Take it easy, it is as simple as this:

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

Observe that I modified the template parameter `TYPE` to **`TYPE&`**, and also added `const` to it. Few or most of the readers would have realized the importance of this change. For those who didn't:

- The `TYPE` type may be large in size, and would demand more space on stack (call-stack). It includes `double` which requires 8 bytes*, some structure or a class, which would demand more bytes to be kept on stack. It essentially means - a new object of given type would be created, copy constructor called, and be put into call stack, followed by destructor call at then end of function.
  Addition of reference (`&`) avoids all this - *reference* of same object is passed.
- Function would not change the argument passed, and therefore addition of `const` to it. It ensures, to the caller of function, that this function (here `PrintTwice`), is not going to change the parameter's value. It also ensures a compiler error if, by mistake, the function itself tries to modify content of (**const**ant) argument.

> * On 32-bit platform, function arguments would require 4-bytes minimum, and in multiple of 4-bytes. This means a `char` or `short` would require 4 bytes in call-stack. An 11-byte object, for example would require 12-bytes in stack.
> Similarly, for 64-bit platform, 8-bytes would be needed. An 11-byte object would require 16-bytes. Argument of type `double` would need 8-bytes.
> All pointers/references takes 4-bytes/8-bytes respectively on 32-bit/64-bit platform, and therefore passing `double` or `double&` would mean the same for 64-bit platform.

And similarly, we should change other function templates as:

```
template<class TYPE>
TYPE Twice(const TYPE& data) // No change for return type
{
   return data * 2;
}

template<class T>
T Add(const T& n1, const T& n2) // No return type change
{
    return n1 + n2;
}

template<class T>
GetAverage(const T tArray[], int nElements)
// GetAverage(const T* tArray, int nElements)
{}
```

Note that it is not possible to have reference and `const` added to return type, unless we intend to return reference (or pointer) of original object that was passed to the function template. The following code exemplifies it:

```
template<class T>
T& GetMax(T& t1, T& t2)
{
    if (t1 > t2)
    {
        return t2;
```

```
    }
    // else
    return t2;
}
```

It is how we utilize return reference:

```
int x = 50;
int y = 64;

// Set the max value to zero (0)
GetMax(x,y) = 0;
```

Note that this is just for illustration, and you would rarely see or write such code. You may, however see such code and may need to write, if the returned object is reference of some UDT. In that case member access operator dot (`.`) or arrow (`->`) would follow the function call. Anyway, this function template returns the *reference* of object which wins the greater-than race. This, definitely, requires `operator >` be defined by type `T`.

You should have noticed, I have not added `const` to any of two parameters passed. This is required; since function returns non-const reference of type `T`. Had it been like:

```
T& GetMax(const T& t1, const T& t2)
```

At the `return` statements, compiler would complain that `t1` or `t2` cannot be converted to non-const. If we add `const` to return type also (`const T& GetMax(...)`), following line at call site would fail to compile:

```
GetMax(x,y) = 0;
```

Since `const` object cannot be modified! You can definitely do forceful const/non-const typecasting, either in function or at call site. But that's a different aspect, a bad design and a non-recommended approach.

Multiple Types with Function Templates

Till now I have covered only one type as template type parameters. With templates, you may have more than one template-type parameters. It goes like:

```
template<class T1, class T2, ... >
```

Where `T1` and `T2` are type-names to the function template. You may use any other specific name, rather than `T1`, `T2`. Note that the usage of '`...`' above does **not** mean that this template specification can take any number of arguments. It is just illustration that template may have any number of arguments.
(As with C++11 standard, templates would allow variable number of arguments - but that thing is out of topic, for now.)

Let's have a simple example taking two template parameters:

```
template<class T1, class T2>
void PrintNumbers(const T1& t1Data, const T2& t2Data)
{
     cout << "First value:" << t1Data;
     cout << "Second value:" << t2Data;
}
```

And we can simply call it as:

```
PrintNumbers(10, 100);    // int, int
PrintNumbers(14, 14.5);   // int, double
PrintNumbers(59.66, 150); // double, int
```

Where each call demands separate template instantiation for the first and second types being passed (or say *inferred*). Therefore, following three function template instances would be populated by compiler:

```
// const and reference removed for simplicity
void PrintNumbers(int t1Data, int t2Data);
void PrintNumbers(int t1Data, double t2Data);
void PrintNumbers(double t1Data, int t2Data);
```

Realize that second and third instantiations are not same, as `T1` and `T2` would infer different data-types (`int`, `double` and `double`,`int`). Compiler will **not** perform any automatic conversion, as it might do for normal function call - A normal function taking `int`, for example, may be passed `short`, or vice-versa. But with templates, if you pass `short` - it is absolutely `short`, not (upgraded to) `int`. So, if you pass (`short`, `int`), (`short`, `short`), (`long`, `int`) - this would result in three different instantiations for `PrintNumbers`!

In similar fashion, function templates may have 3 or more type parameters, and each of them would map to the argument types specified in function call. As an example, the following function template is legal:

```
template<class T1, class T2, class T3>
T2 DoSomething(const T1 tArray[], T2 tDefaultValue, T3& tResult)
{
   ...
}
```

Where `T1` specifies the type of array that would be passed by caller. If array (or pointer) is not passed, compiler will render appropriate error. The type `T2` is used as return type as well as second argument, which is passed by value. Type `T3` is passed as reference (a non-const reference). This function template example given above is just haphazardly chosen, but is a valid function template specification.

By now, I have *stepped-up* and elaborated multiple template parameters. But for a reason, I am *stepping-down* to one parameter function. There is a reason for this, and you will understand it in no time.

Assume that there is a function (**non** templated), which takes an `int` argument:

```
void Show(int nData);
```

And you call it as:

```
Show( 120 );    // 1
Show( 'X' );    // 2
Show( 55.64 );  // 3
```

- Call **1** is perfectly valid since function takes `int` argument, and we are passing `120`.
- Call **2** is valid call since we are passing `char`, which will be promoted by compiler to `int`.
- Call **3** would demand demotion of value - compiler has to convert `double` to `int`, and hence `55` would be passed instead of `55.64`. Yes, this will trigger appropriate compiler warning.

One solution is to modify the function such that it takes `double`, where all three types can be passed. But that wouldn't support all types and may not fit in, or convertible to, `double`. Therefore, you may write set of overloaded functions, taking appropriate types. Armed with knowledge, now, you would appreciate the importance of templates, and would ask to write it a function template instead:

```
template<class Type>
void Show(Type tData) {}
```

Of course, assuming all existing overloads of `Show` were doing the same thing.

Alright, you know this drill. So, what's new in this which caused me to *step-down*?

Well, what if you wanted to pass `int` to function template `Show`, but wish the compiler instantiates as if `double` was passed?

```
// This will produce (instantiate) 'Show(int)'
Show ( 1234 );

// But you want it to produce 'Show(double)'
```

May seem illogical to demand this thing, as of now. But there is valid reason to demand such instantiation, which you will understand and appreciate soon!

Anyway, first see how to demand such absurd thing:

```
Show<double> ( 1234 );
```

Which instantiates the following **template function** (as you know):

```
void Show(double);
```

With this special syntax (`Show<>()`), you are demanding compiler to instantiate `Show` function for the type being explicitly passed, and asking the compiler *not* to deduce type by function argument.

Function Template - Template Function

Important! There is a difference between **function template** and **template function**.

A *function template* is body of a function that is bracketed around `template` keyword, which is not an actual function, and will not be *fully* compiled by compiler, and is not accountable by the linker. At least one call, for particular data-type(s) is needed to instantiate it, and be put into accountability of compiler and linker. Therefore, the instance of function template `Show` is instantiated as `Show(int)` or `Show(double)`.

A *template function*? Simply put, an "instance of a function template", which is produced when you call it, or cause it to get instantiated for particular data type. The instance of function-template is actually a valid function.

An instance of a function template (aka template-function) is not a normal function, under the umbrella of name-decoration system of compiler and linker. That means, an instance of function-template:

```
template<class T>
void Show(T data)
{ }
```

for template argument `double`, it is **not**:

```
void Show(double data){}
```

but actually:

```
void Show<double>(double x){}
```

For long, I did not uncover this, only for simplicity, and now you know it! Use your compiler/debugger to find out the actual instantiation of a function template, and see the full prototype of a function in call-stack or generated code.

And hence, now you know the mapping between these two:

```
Show<double>(1234);
...
void Show<double>(double data); // Note that data=1234.00, in this case!
```

Explicit Template Argument Specification

Stepping back (up) to the multiple template argument discussion.

We have following function template:

```
template<class T1, class T2>
void PrintNumbers(const T1& t1Data, const T2& t2Data)
{}
```

And have following function calls, causing 3 different instances of this function template:

```
PrintNumbers(10, 100);    // int, int
PrintNumbers(14, 14.5);   // int, double
PrintNumbers(59.66, 150); // double, int
```

And what if you just needed only one instance - both arguments taking `double`? Yes, you are willing to pass `int`s and let them be promoted `double`s. Coupled with the understanding you just gained, you would call this function-template as:

```
PrintNumbers<double, double>(10, 100);    // int, int
PrintNumbers<double, double>(14, 14.5);   // int, double
PrintNumbers<double, double>(59.66, 150); // double, int
```

Which would produce only the following *template function:*

```
void PrintNumbers<double, double>(const double& t1Data, const T2& t2Data)
{}
```

And the concept of passing template type parameters this way, from the call-site, is known as **Explicit Template Argument Specification.**

Why would you need explicit type specification? Well, there are multiple reasons:

- You want only specific type to be passed, and not let the compiler *intelligently deduce* one or more template argument types solely by the actual arguments (function parameters).

For example, there is one function template, `max`, taking **two** arguments (via only **one** template type parameter):

```
template<class T>
T max(T t1, T t2)
{
   if (t1 > t2)
      return t1;
   return t2;
}
```

And you attempt to call it as:

```
max(120, 14.55);
```

It would cause a compiler error, mentioning there is an ambiguity with template-type `T`. You are asking the compiler to deduce one type, from two types! One solution is to change `max` template so that it takes two template parameters - but you aren't author of that function template.

There you use explicit argument specification:

```
max<double>(120, 14.55); // Instantiates max<double>(double,double);
```

Undoubtedly notice and understand that I have passed explicit specification only for **first** template parameter, the second type is deduced from second argument of function call.

- When function-template takes template-type, but not from its function arguments.

A simple example:

```
template<class T>
void PrintSize()
{
   cout << "Size of this type:" << sizeof(T);
}
```

You cannot call such function template simply as:

```
PrintSize();
```

Since this function template would require template type argument specification, and it cannot be deduced automatically by compiler. The correct call would be:

```
PrintSize<float>();
```

which would instantiate `PrintSize` with `float` template argument.

- When function template has a return type, which cannot be deduced from arguments, or when function template doesn't have any argument.

An example:

```
template<class T>
T SumOfNumbers(int a, int b)
{
   T t = T(); // Call default CTOR for T

   t = T(a)+b;


   return t;
}
```

Which takes two `int`s, and sums them up. Though, summing them in `int` itself is appropriate, this function template gives *opportunity* to calculate the sum (using `operator+`) in any type as required by caller. For example, the get the result in `double`, you would call it as:

```
double nSum;
nSum = SumOfNumbers<double>(120,200);
```

The last two are just simplified examples for completeness, just to give you the *hint* where Explicit Template Argument Specification would fit in. There are more concrete scenarios where this *explicitness* would be needed, and would be covered in next part.

Default Arguments with Function Templates

For readers, who *do* know about default template type specification in template' arena - this is **not** about default template-type arguments. Default template-types, is anyway, not allowed with function-templates. For readers, who do *not* know about it, do not worry - this paragraph is not about default template type specification.

As you know, a C++ function may have default arguments. The default-ness may only go from right to left, meaning, if **nth** argument is required to be default, (**n+1**)**th** must also be default, and so on till last argument of function.

A simple example to explicate this:

```
template<class T>
void PrintNumbers(T array[], int array_size, T filter = T())
{
   for(int nIndex = 0; nIndex < array_size; ++nIndex)
   {
       if ( array[nIndex] != filter) // Print if not filtered
           cout << array[nIndex];
   }
}
```

This function template would print, as you can guess, all numbers except which are filtered out by third argument: `filter`. The last, optional function argument, is defaulted to default-value of type `T`, which, for all basic type would mean zero. Therefore, when you call it as:

```
int Array[10] = {1,2,0,3,4,2,5,6,0,7};
PrintNumbers(Array, 10);
```

It would be instantiated as:

```
void PrintNumbers(int array[], int array_size, int filter = int())
{}
```

The `filter` argument would be rendered as: `int filter = 0`.

As obvious, when you call it as:

```
PrintNumbers(Array, 10, 2);
```

The third argument gets value `2`, and not the default value `0`.

It should be clearly understood that:

- Type `T` must have default constructor available. And of course, all operators as may be required by function body, for type `T`.
- The default argument must be *deducible* from the other non-default types the template takes. In `PrintNumbers` example, type of `array` would facilitate deduction for `filter`.
  If not, you must use explicit template argument specification to specify type of default argument.

For sure, the default argument may not necessarily be default value for type `T` (pardon the pun). It means, the default-argument may not always need to depend on default-constructor of type `T`:

```
template<class T>
void PrintNumbers(T array[], int array_size, T filter = T(60))
```

Here, the default function argument doesn't use default-value for type `T`. Instead, it uses value `60`. This, for sure, requires the type `T` to have copy-constructor which accepts an `int` (for `60`).

Finally, here comes an end to 'Function Templates' for this part of article. I assume you enjoyed reading and grasping these *basics* of function templates. Next part would cover more intriguing aspects of Template programming.

Class Templates

More often, you would design and use class templates, than function templates. In general, you use a class template to define an abstract type whose behavior is generic and is reusable, adaptable. While some text would start by giving example about data-structures like linked-lists, stacks, queues and similar *containers*. I would start with very basic examples, that are easy to understand.

Let's take a simple class, which sets, gets and prints the value stored:

```
class Item
{
    int Data;
public:
    Item() : Data(0)
    {}

    void SetData(int nValue)
    {
        Data = nValue;
    }

    int GetData() const
    {
        return Data;
    }

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

One constructor which initializes `Data` to `0`, Set and Get methods, and a method to print current value. Usage is also quite simple:

```
Item item1;
item1.SetData(120);
item1.PrintData(); // Shows 120
```

Nothing new for you, for sure! But when you need similar abstraction for other data-type, you need to duplicate code of entire class (or at least the required methods). It incurs code maintenance issues, increases code size at source code as well as at binary level.

Yes, I can sense your intelligence that I am going to mention C++ templates! The templated version of the same class in form of **class template** is as below:

```
template<class 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 class template declaration starts with same syntax as function templates:

```
template<class T>
class Item
```

Note that the keyword `class` is used two times - firstly to specify template type specification (`T`), and secondly to specify that this is a C++ class declaration.

To completely turn `Item` into a class template, I replaced all instances of `int` with `T`. I also used `T()` syntax to call default constructor of `T`, instead of hard-coded `0` (zero), in the constructor's initializer' list. If you've read function templates section completely, you know the reason!

And usage as also quite simple:

```
Item<int> item1;
item1.SetData(120);
item1.PrintData();
```

Unlike function template instantiation, where arguments of function itself helps the compiler to deduce template type arguments, with class templates you must explicitly pass template type (in angle brackets).

The code snippet shown above causes class template `Item` to instantiate as `Item<int>`. When you create another object with different type using `Item` class template as:

```
Item<float> item2;
float n = item2.GetData();
```

It would cause `Item<float>` to get instantiated. It is important to know that there is absolutely no relation between two instantiations of class template - `Item<int>` and `Item<float>`. For the compiler and linker, these two are different entities - or say, different classes.

First instantiation with type `int` produces following methods:

- `Item<int>::Item()` constructor
- `SetData` and `PrintData` methods for type `int`

Similarly, second instantiation with type `float` would produce:

- `Item<float>::Item()` constructor
- `GetData` method for `float` type

As you know `Item<int>` and `Item<float>` are two different classes/types; and therefore, following code will not work:

```
item1 = item2; // ERROR : Item<float> to Item<int>
```

Since both types are different, the compiler will not call *possible* default assignment operator. Had `item1` and `item2` were of same types (say both of `Item<int>`), the compiler would happily call assignment operator. Though, for the compiler, conversion between `int` and `float` is possible, it is not possible for different UDT conversions, even if underlying data members are same - this is simple C++ rule.

At this point, clearly understand that only following set of methods would get instantiated:

- `Item<int>::Item()` - constructor
- `void Item<int>::SetData(int)` method
- `void Item<int>::PrintData() const` method
- `Item<float>::Item()` - constructor
- `float Item<float>::GetData() const` method

The following methods will **not** get second phase compilation:

- `int Item<int>::GetData() const`
- `void Item<float>::SetData(float)`
- `void Item<float>::PrintData() const`

Now, what is a second phase compilation? Well, as I already elaborated that template-code would be compiled for basic syntax checks, irrespective of it being called/instantiated or not. This is known as first-phase compilation.

When you actually call, or somehow trigger it to be called, the function/method for the particular type(s) - then only it gets special treatment of *second-phase* compilation. Only through the second phase compilation, the code actually gets fully-compiled, against the type for which it is being instantiated.

Though, I could have elaborated this earlier, but this place is appropriate. How do you find out if function is getting first-phase and/or second-phase compilation?

Let's do something weird:

```
T GetData() const
 {
  for())

  return Data;
 }
```

There is an extra parenthesis at the end of `for` - which is incorrect. When you compile it, you would get host of errors, **irrespective** of it being called or not. I have checked it using Visual C++ and GCC compilers, and both complain. This validates first-phase compilation.

Let's slightly change this to:

```
T GetData() const
{
  T temp = Data[0]; // Index access ?
  return Data;
}
```

Now compile it **without** calling `GetData` method for any type - and there won't be any quench from the compiler. This means, at this point, this function doesn't get phase-two compilation treatment!

As soon as you call:

```
Item<double> item3;
item2.GetData();
```

you would get error from compiler that `Data` is not an array or pointer, which could have `operartor []` attached to it. It proves that only selected functions would get special privilege of phase-two compilation. And this phase-two compilation would happen separately for all unique types you instantiate class/function template for.

One interesting thing, you can do is:

```
T GetData() const
{
  return Data % 10;
}
```

Which would get successfully compiled for `Item<int>`, but would fail for `Item<float>`:

```
item1.GetData(); // item1 is Item<int>

// ERROR
item2.GetData(); // item2 is Item<float>
```

Since `operator %` is not applicable for `float` type. Isn't it interesting?

Multiple Types with Class Templates

Our first class-template `Item` had only one template type. Now let's construct a class that would have two template-type arguments. Again, there could have been somewhat complex class template example, I would like to keep it simple.

At times, you do require some native structure to keep few data members. Crafting a unique `struct` for the same appears somewhat needless and unnecessary-work. You would soon come out of names for different structures having few members in it. Also, it increases code length. Whatever your perspective may be for this, I am using it as an example, and deriving a class template having two members in it.

STL programmers would find this as equivalent to `std::pair` class template.

Assume you have a structure `Point`,

```
struct Point
{
    int x;
    int y;
};
```

which is having two data-members. Further, you may also have another structure `Money`:

```
struct Money
{
    int Dollars;
    int Cents;
};
```

Both of these structures have almost similar data-members in it. Instead of re-writing different structures, wouldn't it be better to have it at one place, which would also facilitate:

- Constructor having one or two arguments of given types, and a copy-constructor.
- Methods to compare two objects of same type.
- Swapping between two types
- And more.

You might say you can use inheritance model, where you'd define all required methods and let derive class customize it. Does it fit in? What about the data-types you chosen? It might be `int`, `string`, or `float`, *some-class* as types. In short, inheritance will only complicate the design, and will not allow plug-in feature facilitated by C++ templates.

There we use class templates! Just define a class *template* for two types, having all required methods. Let's start!

```
template<class Type1, class Type2>
struct Pair
{
    // In public area, since we want the client to use them directly.
    Type1 first;
    Type2 second;
};
```

Now, we can use `Pair` class template to *derive* any type having two members. An example:

```
// Assume as Point struct
Pair<int,int> point1;

// Logically same as X and Y members
point1.first = 10;
point1.second = 20;
```

Understand that type of `first` and `second` are now `int` and `int`, respectively. This is because we instantiated `Pair` with these types.

When we instantiate it like:

```
Pair<int, double> SqRoot;

SqRoot.first = 90;
SqRoot.second = 9.4868329;
```

`first` would be of `int` type, and `second` would be of `double` type. Clearly understand that `first` and `second` are data-members, and not functions, and therefore there is no runtime penalty of *assumed* function call.

Note: In this part of article, all definitions are within the class declaration body only. In next part, I would explain how to implement methods in separate implementation file, and issues related with that. Therefore, all method definitions shown should be assumed within `class ClassName{...};` only.

The following given default constructor would initialize both members to their default values, as per data type of `Type1` and `Type2`:

```
Pair() : first(Type1()), second(Type2())
{}
```

Following is a parameterized constructor taking `Type1` and `Type2` to initialize values of `first` and `second`:

```
Pair(const Type1& t1, const Type2& t2) :
  first(t1), second(t2)
  {}
```

Following is a copy-constructor which would copy one `Pair` object from another `Pair` object, of exactly same type:

```
Pair(const Pair<Type1, Type2>& OtherPair) :
  first(OtherPair.first),
  second(OtherPair.second)
 {}
```

Please note that it is very much required to specify template type arguments of `Pair<>`, for the argument of this copy-constructor. The following specification wouldn't make sense, since `Pair` is *not* a non-template type:

```
Pair(const Pair& OtherPair) // ERROR: Pair requires template-types
```

And here is an example using parametrized constructor and copy-constructor:

```
Pair<int,int> point1(12,40);
Pair<int,int> point2(point1);
```

It is important to note that if you change any of the template type parameters of either of the objects, `point2` or `point1`, you wouldn't be able to copy-construct it using `point1` object. Following would be an error:

```
Pair<int,float> point2(point1);  // ERROR: Different types, no conversion possible.
```

Though, there is a possible conversion between `float` to `int`, but there is no possible conversion between `Pair<int,float>` to `Pair<int,int>`. The copy constructor cannot take other *type* as copyable object. There is a solution to this, but I would discuss it in next part.

In similar fashion, you can implement comparison operators to compare two objects of same `Pair` type. Following is an implementation of equal-to operator:

```
bool operator == (const Pair<Type1, Type2>& Other) const
{
  return first == Other.first &&
         second == Other.second;
}
```

Note that I used `const` attribute, for argument and for the method itself. Please fully understand the first line of above' method definition!

Just like copy-constructor call, you must pass exactly the same type to this comparison operator - compiler will not attempt to convert different `Pair` types. An example:

```
if (point1 == point2) // Both objects must be of same type.

   ...
```

For a solid understanding for the concepts covered till here, please implement following methods by on your own:

- All remaining 5 relational operators
- Assignment operator
- `Swap` method
- Modify both constructors (except copy-constructor), and combine them into one so that they take both parameters as default. This means, implement only one constructor that can take 0,1 or 2 arguments.

`Pair` class is an example for two types, and it can be used instead of defining multiple structures having just two data-members. The drawback is just with remembering what `first` and `second` would mean (X or Y?). But when you well-define a template instantiation, you would always know and use `first` and `second` members appropriately.

Ignoring this one disadvantage, you would achieve all the features in the *instantiated* type: constructors, copy-constructor, comparison operators, swap method etc. And, you'd get all this without re-writing the required code for various two-member structures you would need. Furthermore, as you know, only the set of *required* methods would get compiled and linked. A bug fix in class template would automatically be reflected to all instantiations. Yes, a slight modification to class template may also raise bunch of errors, of other types, if the modification fails to comply with existing usage.

Likewise, you can have a class template `tuple` which allows three (or more) data-members. Please try to implement class `tuple` with three members (`first`, `second`, `third`) by yourself:

```
template<class T1, class T2, class T3>
class tuple
```

Non-type Template Arguments

Alright, we have seen that class templates, just like function templates, can take multiple type arguments. But class templates also allow few non-type template arguments. In this part, I will elaborate only one non-type: **integer**.

Yes, a class template may take a integer as template argument. First a sample:

```
template<class T, int SIZE>
class Array{};
```

In this class template declaration, `int SIZE` is a non-type argument, which is an integer.

- Only integral data-types can be non-type integer argument, it includes `int`, `char`, `long`, `long long`, *`unsigned`* variants and `enum`s. Types such as `float` and `double` are not allowed.
- When being instantiated, only compile time constant integer can be passed. This means `100`, `100+99`, `1<<3` etc are allowed, since they are compiled time constant expressions. Arguments, that involve function call, like `abs(-120)`, are not allowed.
  As a template argument, floats/doubles etc may be allowed, if they can be converted to integer.

Fine. We can instantiate class template `Array` as:

```
Array<int, 10> my_array;
```

So what? What's the purpose of `SIZE` argument?

Well, within the class template you can use this non-type integer argument, wherever you could have used an integer. It includes:

- Assigning static const data-member of a class.

```
template<class T, int SIZE>
class Array
{
 static const int Elements_2x = SIZE * 2;
};
```

[First two lines of class declaration will not be shown further, assume everything is within class' body.]

Since it is allowed to initialize a *static-constant-integer* within class declaration, we can use non-type integer argument.

- To specify default value for a method.
  (Though, C++ also allows any non-constant to be a default parameter of a function, I have pointed this one for just for illustration.)

```
void DoSomething(int arg = SIZE);
// Non-const can also appear as default-argument...
```

- To define the size of an array.

This one is important, and non-type integer argument is often used for this purpose. So, let's implement the class template `Array` utilizing `SIZE` argument.

```
private:
   T TheArray[SIZE];
```

`T` is the type of array, `SIZE` is the size (integer) - as simple as that. Since the array is in private area of class, we need define several methods/operators.

```
// Initialize with default (i.e. 0 for int)
void Initialize()
{
    for(int nIndex = 0; nIndex < SIZE; ++nIndex)
        TheArray[nIndex] = T();
}
```

For sure, the type `T` must have a default constructor and an assignment operator. I will cover these things (*requirements*) for function template and class templates in next part.

We also need to implement array-element access operators. One of the overloaded index-access operator sets, and the other one gets the value (of type `T`):

```
T operator[](int nIndex) const
{
  if (nIndex>0 && nIndex<SIZE)
  {
    return TheArray[nIndex];
  }
  return T();
}

T& operator[](int nIndex)
{
   return TheArray[nIndex];
}
```

Note that first overload (which is declared `const`) is get/read method, and has a check to see if index is valid or not, otherwise returns default value for type `T.`

The second overload returns the **reference** of an element, which can be modified by caller. There is no index validity check, since it has to return a reference, and therefore local-object (`T()`) cannot be returned. You may, however, check the index argument, return default value, use *assert*ion and/or throw an exception.

Let's define another method, which would *logically* sum all the elements of `Array`:

```
T Accumulate() const
{
  T sum = T();
  for(int nIndex = 0; nIndex < SIZE; ++nIndex)
  {
     sum += TheArray[nIndex];
  }
  return sum;
}
```

As you can interpret, it requires `operator +=` to be available for target type `T`. Also note that return type is `T` itself, which is appropriate. So, when instantiate `Array` with some string class, it will call `+=` on each iteration and would return the combined string. If target type doesn't have this `+=` operator defined, and you call this method, there would be an error. In that case, you either - don't call it; or implement the required operator overload in the target class.

Template Class as Argument to Class Template

While it is a vague statement to understand, and invites some ambiguities, I would attempt my best to remove the fogginess.

First, recollect the difference between *template-function* and a *function-template*. If the *neurons* have helped to transfer the correct information to the *cache* of your brain-box, you now *callback* that template-function is an instance of function-template. If search-subsystem of your brain is not responding, please reload the information again!

An instance of **class template** is **template class.** Therefore, for following class template:

```
template<class T1, class T2>
class Pair{};
```

The instantiation of this template is a template-class:

```
Pair<int,int> IntPair;
```

Clearly understand that `IntPair` is **not** a template-class, is **not** instantiation for class template. It is an *object* of a particular instantiation/class-template. The template-class/instantiation is `Pair<int,int>`, which produces another class type (compiler, our friend does this, you know!). Essentially this is what template-class would be produced by compiler for this case:

```
class Pair<int,int>{};
```

There is more precise definition for a template-class, select this single line of code for easy understanding. Detailed explication would come in next installment of this series.

Now, let's come to the point. What if you pass a template-class to some class-template? I mean, what does it mean by following statement?

```
Pair<int, Pair<int,int> > PairOfPair;
```

Is it valid - if so, what does it mean?
Firstly, it is perfectly valid. Secondly, it instantiates **two** template classes:

- **`Pair<int,int>`** - **A**
- **`Pair<int, Pair<int,int> >`** -- **B**

Both **A** and **B** types would be instantiated by compiler, and if there is any error, arising due to any type of these two template classes, compiler would report. To simplify this *complex* instantiation, you may do:

```
typedef Pair<int,int> IntIntPair;
...
Pair<int, IntIntPair> PairOfPair;
```

You can assign `first` and `second` members of `PairOfPair` object like this:

```
PairOfPair.first = 10;
PairOfPair.second.first = 10;
PairOfPair.second.second= 30;
```

Note that `second` member in last two lines is of type `Pair<int,int>`, and therefore it has same set of members to be accessed further. That's the reason `first` and `second` members can be used, in *cascaded* manner.

Now you (hopefully) understand that class template (`Pair`) is taking template-class (`Pair<int,int>`) as argument and inducing the final instantiation!

An interesting instantiation, in this discussion, would be of `Array` with `Pair`! You know that `Pair` takes two template type arguments, and `Array` takes one type argument, and a size (integer) argument.

```
Array< Pair<int, double>, 40> ArrayOfPair;
```

Here `int` and `double` are type-arguments for `Pair`. Hence, the first template-type of `Array` (marked bold) is `Pair<int,double>`. The second argument is constant `40`. Can you answer this: Would the constructor of `Pair<int,double>` be called? When it will be called? Before you answer that, I just reverse the instantiation as:

```
Pair<int, Array<double, 50>> PairOfArray;
```

Wohoo! What does it mean?
Well, it means: `PairOfArray` is an instantiation of `Pair`, which is taking first type as `int` (for `first` member), and second type (`second`) is an `Array`. Where `Array` (the second type of `Pair`) is `50` elements of type `double`!

Don't kill me for this! Slowly and clearly understand these basic concepts of templates. Once you get the crystal-clear understanding, you would *love* templates!

Here again, I used a template-class (`Array<double,50>`) as an argument to an instance of other type (`Pair<int,...>`).

Okay, but what right-shift operator (**`>>`**) is doing above? Well, that's not an operator, but just ending of `Array`'s type specification, followed by ending of `Pair` type specification. Some old compilers required us to put a space in between two greater-than symbols, so to avoid error or confusion.

```
Pair<int, Array<double, 50>  > PairOfArray;
```

At present, almost all modern C++ compilers are smart enough to understand that this is used to end template type specification, and therefore you need not to worry. Therefore, you may freely use two or more **`>`** symbols to end template specification(s).

Kindly note that passing a template class (instantiation) is nothing very specific in C++ terms - it is just a type that a class template would take.

Finally, Here I put usage examples both objects. First the constructors.

```
Array< Pair<int, double>, 40> ArrayOfPair;
```

This will cause the constructor of `Pair` to be called **40** times, since there is declaration of constant-size array in `Array` class template:

```
T TheArray[SIZE];
```

Which would mean:

```
Pair<int,double> TheArray[40];
```

And hence the required number of calls to constructor of `Pair`.

For the following object construction:

```
Pair<int, Array<double, 50>> PairOfArray;
```

The constructor of `Pair` would initialize first argument with `0` (using `int()` notation), and would call constructor of `Array` with `Array()` notation, as shown below:

```
Pair() : first(int()), second(Array())
 {}
```

Since the default constructor of `Array` class template is provided by compiler, it would be called. If you don't understand the stuff written here, please sharpen your C++ skills.

Assigning one element of `ArrayOfPair`:

```
ArrayOfPair[0] = Pair<int,double>(40, 3.14159);
```

Here, you are calling non-const version of `Array::operator[]`, which would return the *reference* of first element of `Array` (from `TheArray`). The element, as you know, is of type `Pair<int,double>`. The expression on right side of assignment operator is just calling the constructor for `Pair<int,double>` and passing required two arguments. The assignment is done!

Default Template Arguments with Class Templates

First let me eliminate any ambiguity with 'Default Argument' phrase. The same phrase was used in Function Template section. In that sub-section, the default-argument referred to arguments of function parameters itself, not the type-arguments of function template. Function templates, anyway, do **not** support default arguments for template-arguments. As a side note, please know that methods of a class template can take default arguments, as any ordinary function/method would take.

Class templates, on other hand, do support default-argument for the type/non-type arguments for the template parameters. Throwing you an example:

```
template<class T, int SIZE=100>
class Array
{
private:
   T TheArray[SIZE];
   ...
};
```

I have just modified around `SIZE` in first line for class template `Array`. The second template parameter, an integer constant specification, is now set to `100`. It means, when you use it in following manner:

```
Array<int> IntArray;
```

It would essentially mean:

```
Array<int, 100> IntArray;
```

Which would be automatically placed by compiler during the instantiation of this class template. Of course, you can specify custom array size by explicitly passing the second template argument:

```
Array<int, 200> IntArray;
```

Do remember that when you explicitly pass the default argument's parameter, with the same argument specified in class template declaration, it would instantiate it only once. By this, I mean, the following two objects created would instantiate only one class: `Array<int,100>`

```
Array<int> Array1;
Array<int,100> Array2;
```

Of course, if you change the default' template parameter in class template definition, with value other than `100`, it would cause two template instantiations, since they would be different types.

You can customize the default argument by using `const` or `#define`:

```
const int _size = 120;
// #define _size 150
template<class T, int SIZE=_size>
class Array
```

For sure, using `_size` symbol instead of hard-coded constant value mean the same. But using a symbol would ease the default' specification. Irrespective of how you specify the default template parameter for integer (which is a non-type template argument), it must be a compile time constant expression.

You would generally *not* use default specification for non-type integer parameter, unless you are utilizing templates for advanced stuff, like meta-programming, static-asserts, SFINAE etc, which definitely demands a separate part. More often you would see and implement default parameters for class templates, which are **data-types**. An example:

```
template<class T = int>
class Array100
{
    T TheArray[100];
};
```

It defines an array of type `T` of size `100`. Here, the type argument is defaulted to **`int`**. That means, if you don't specify the type while instantiating `Array100`, it would map to `int`. Following is an example on how to use it:

```
Array100<float> FloatArray;
Array100<> IntArray;
```

In the first instantiation, I passed `float` as template type, and in second call I kept it default (to `int`), by using **`<>`** notation. While there are more uses of this notation in template programming, which I would cover up in later parts, it is very much required for this case also. If you try to use the class template as:

```
Array100 IntArray;
```

It would result in compiler errors, saying `Array100` requires template parameters. Therefore, you must use empty-set of angle-brackets (`<>`) to instantiate a class template, if all template arguments are default, and you wish to use defaults.

Important thing to remember is that a non-template class of name `Array100` will **not** be allowed also. Definition of a non-template class like as given below, along with template class (above or below each other), will upset the compiler:

```
class Array100{}; // Array100 demands template arguments!
```

Now, let's mix both type and non-type argument in our class `Array`:

```
template<class T = int, int SIZE=100>
class Array
{
```

```
    T TheArray[SIZE];
    ...
};
```

Finally, both type and the size arguments are marked default with `int` and `100` respectively. Clearly understand that first `int` is for default specification of `T`, and second `int` is for non-template constant specification. For simplicity and better readability, you should keep them in different lines:

```
template<class T = int,
         int SIZE=100>
class Array{};
```

Now, use use your intelligence to parse the meaning of following instantiations:

```
Array<>            IntArray1;
Array<int>         IntArray2;
Array<float, 40>   FlaotArray3;
```

Just like explicit specification in function templates, specifying only the trailing template arguments is not allowed. Following is an error:

```
Array<, 400> IntArrayOf500; // ERROR
```

As a final note, do remember that following two object creations would instantiate only one class template, since essentially they are exactly the same:

```
Array<>          IntArray1;
Array<int>       IntArray2
Array<int, 100>  IntArray3;
```

Defaulting a template type on another type

It is also possible to default a type/non-type parameter on a previously arrived template parameter. For instance, we can modify the `Pair` class so that second type would be same as first type, if second type is not explicitly specified.

```
template<class Type1, class Type2 = Type1>
class Pair
{
    Type1 first;
    Type2 second;
};
```

In this modified class template `Pair`, `Type2` now defaults to `Type1` type. An instantiation example:

```
Pair<int> IntPair;
```

Which, as you can guess, is same as:

```
Pair<int,int> IntPair;
```

But saves you from typing the second parameter. It is also possible to let the first argument of `Pair` be default also:

```
template<class Type1=int, class Type2 = Type1>
class Pair
{
    Type1 first;
    Type2 second;
};
```

Which means, if you don't pass any template argument, `Type1` would be `int`, and hence `Type2` would also be `int` !

The following usage:

```
Pair<> IntPair;
```

Instantiates following class:

```
class Pair<int,int>{};
```

For sure, it is also possible to default non-type arguments on another non-type argument. An example:

```
template<class T, int ROWS = 8, int COLUMNS = ROWS>
class Matrix
{
    T TheMatrix[ROWS][COLUMNS];
};
```

But, the *dependent* template parameter must be on the *right* of which it is dependent upon. Following would cause errors:

```
template<class Type1=Type2, class Type2 = int>
class Pair{};

template<class T, int ROWS = COLUMNS, int COLUMNS = 8>
class Matrix
```

Class' Methods as Function Templates

Though, this one isn't for absolute beginners, but since I have covered both function templates and class templates - the elaboration of this concept is logical for this first part of this series itself.

Consider a simple example:

```
class IntArray
{
    int TheArray[10];
public:
    template<typename T>
    void Copy(T target_array[10])
    {
       for(int nIndex = 0; nIndex<10; ++nIndex)
       {
          target_array[nIndex] = TheArray[nIndex];
          // Better approach:
            //target_array[nIndex] = static_cast<T>(TheArray[nIndex]);
       }
    }
};
```

The class `IntArray` is simple, non-template class, having an integer array of `10` elements. But the method **`Copy`** is a designed as a function template (method template?). It takes one template type parameter, which would be deduced by compiler automatically. Here is how we can use it:

```
IntArray int_array;
float float_array[10];

int_array.Copy(float_array);
```

As you can guess, `IntArray::Copy` would be instantiated with type `float`, since we are passing float-array to it. To avoid confusion and understand it better, just think of `int_array.Copy` as `Copy` only, and `IntArray::Copy<float>(..)` as `Copy<float>(..)` only. The *method* template of a class is nothing but an ordinary function template embedded in a class.

Do notice that I used `10` as array size everywhere. Interestingly, we can also modify the class as:

```
template<int ARRAY_SIZE>
class IntArray
{
    int TheArray[ARRAY_SIZE];
public:
    template<typename T>
    void Copy(T target_array[ARRAY_SIZE])
    {
       for(int nIndex = 0; nIndex<ARRAY_SIZE; ++nIndex)
       {
            target_array[nIndex] = static_cast<T>(TheArray[nIndex]);
       }
    }
};
```

Which makes the class `IntArray` and the method `Copy`, better candidates to be in the realm of template programming!

As you would have intelligently guessed, `Copy` method is nothing but an array-conversion routine, which converts from `int` to any type, wherever conversion from `int` to given type is possible. This is one of the valid case where class method can be written as function templates, taking template arguments by themselves. Please modify this *class template* so that it can work for any type of array, not just `int.`

For sure, 'explicit template argument specification' with method template is also possible. Consider another example:

```
template<class T>
class Convert
{
   T data;
public:
   Convert(const T& tData = T()) : data(tData)
   { }

   template<class C>
   bool IsEqualTo( const C& other ) const
   {
       return data == other;
   }
};
```

Which can be utilized as:

```
Convert<int> Data;
float Data2 = 1 ;

bool b = Data.IsEqualTo(Data2);
```

It instantiates `Convert::IsEqualTo` with `float` parameter. Explicit specification, as given below, would instantiate it with `double`:

```
bool b = Data.IsEqualTo<double>(Data2);
```

One of the astounding thing, with the help of templates, you can do it by defining conversion operator on top of template!

```
template<class T>
operator T() const
{
    return data;
}
```

It would make possible to convert the `Convert`' class template instance into any type, whenever possible. Consider following usage example:

```
Convert<int> IntData(40);
float FloatData;
```

```
double DoubleData;


FloatData = IntData;
DoubleData = IntData;
```

Which would instantiate following two methods (fully qualified names):

```
Convert<int>::operator<float> float();
Convert<int>::operator<double> double();
```

On one hand it provides good flexibility, since without writing extra code, `Convert` can convert itself (the specific instantiation) to any data-type - whenever conversion is possible at compilation level. If conversion is not possible, like from `double` to string-type, it would raise an error.

But on the other hand, it also invites trouble by possibility of inadvertently inserting bugs. You may not want to have conversion operator called, and it is called (compiler code generated) without you knowing about it.

At The End

You have just seen slight glimpse of power and flexibility provided by templates. Next part will cover more of advanced and intriguing concepts. My humble and aspiring request to all readers is to *play* more and more with templates. Try to gain firm understanding on one aspect first (like function template only), rather than just hastily jumping to other concept. Initially do it with your ***test*** projects/code-base, and not with any existing/working/production code.

Following is a summary of what we have covered:

- To avoid unnecessary code duplication and code maintenance issues, specially when code is exactly same, we can use templates. Templates are far better approach than using C/C++ macros or functions/classes running on top of void-pointers.
- Templates are not only type-safe, but also reduces unnecessary code-bloat which would not be referred (not generated by compiler).
- Function templates are used to put a code that is not a part of class, and is same/almost-same for different data-types. At most of the places, compiler would automatically determine the type. Otherwise you have to specify the type, or you may also specify explicit type yourself.
- Class templates makes it possible to wrap any data type around specific implementation. It may be an array, string, queue, linked-list, thread-safe *atomic* implementation etc. Class templates do facilitate default template type specification, which function template don't support.

Hope you have enjoyed the article, and cleared out the mental-block that templates are complicated, unnecessarily, bizarre. Second part would be arriving soon!

History

- First release: October 8, 2011 - *Covered basics of templates in C++, function templates.*
- First amendment: Oct 9, 2011 - *Class Templates and multiple types in class templates, non-template type*
- Third amendment: Oct 11, 2011 - *Template class as argument to class template*
- Fourth amendment: Oct 12, 2011 - *Default Template Arguments with Class Templates*
- Fifth amendment: Oct 13, 2011 - Methods as Function Templates, finishing lines.
- Sixth amendment: Mar 8, 2012 - Basic corrections.

License

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

128 messages have been posted for this article Visit **https://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part-1** to post and view comments on this article, or click **here** to get a print view with messages.