# C++ Templates Explained In Color

REVISION 0

Hawthorne-Press.com

C++ Templates Explained In Color

Published by

Hawthorne-Press.com
916 Adele Street
Houston, Texas 77009, USA

# Table of Contents

# C++ Templates Explained In Color

## Introduction

Many concepts in C++ are relatively easy to explain to an experienced developer and they gravitate to using them quite easily. Prime examples are STL Templates. A developer will readily adapt to the concept of a Container Class Template such as *List* or *Vector* and use them extensively.

Where many programmers fail to adapt is when they define their own classes and functions. Most textbooks ***academically*** define Template generation descriptions. This document will use ***color*** to make such descriptions easier to comprehend.

Many sources provide deeper discussion of Templates. *The purpose of the document is to illustrate the mechanics of declaring Templates easier to comprehend*. Color will identify the various portions of a Template Declaration. Use this document as a supplement to Template Discussions from other sources.

## Document Structure

As stated above, we will explore C++ Templates and use color to make them more understandable. This discussion will be broken into the following sections:

1. Converting a Normal Class into a Template Class
2. Converting a Function into a Template Function
3. Template Type and Non-Type Paramters
4. Template Functions containing Casts
5. Template Specialization
6. Template Classes containing Template Functions

## Document Color Coding Standard

Understanding the way colors are used will help developers grasp these C++ concepts more easily. Additionally, all code will be shown in boxed source code style (Courier New).

- **Red** – Shows Template Declarations: `template <class RepType> …`
- **Green** – Indicates Type Substitutions: `RepType add(RepType x , RepType y);`
- **Orange –** Secondary Type Substitutions**:** `template <class RepType, class RepType2>…`
- **Black** –Non Template Code Examples `class int_calc { … };`
- **Blue –** Instantiated Code Examples: `int_calc int_calc_obj;`
- **Purple –** Working Code Examples**:** **Each section has a working program example.**

## Example Source Code

The example source code associated with the five Template code examples is available as a downloadable ***Tar*** file available at www.hawthorne-press.com.

Download Filename: ***cpp_example_source.tgz***

# Section 1: Converting a Normal Class into a Template Class

First, let us define a normal Integer Class containing two member functions, *add,* and *multiply*. This Class, as defined, should only be used for Integer arguments and an Integer result. Nevertheless, as noted later, it can be abused!

## Example of a Normal Integer Class Definition

```
class  int_calc
{
public:
    bool less_than(int x, int y)   // Member Function defined in line
    {
        return(x < y);
    }
    int multiply(int x, int y)     // Member defined in-line
    {
        return x*y;
    }
    int add(int x, int y);         // Member ProtoRepType Only
};

// Member Function Body Defined Outside the Class!
int int_calc::add(int x, int y)
{
    return   x + y;
};
```

### Instantiating an Integer Class

After ***int_class*** is instantiated it can used to create ***int_class*** Objects. As a matter of programming style, it should be used to manipulate Integer Objects.

```
int_calc  int_calc_obj;

bool result1 = int_calc_obj.less_than(3, 4)     // bresult = TRUE
int result2   = int_calc_obj.add(3, 4);         // result = 7
int result3   = int_calc_obj.multiply(5, 10);   // result = 50
```

We will cover ***explicit casts*** later in this document. However, be aware that the ***implicit casts*** can occur.

### Abusing an Integer Class

While the following code is legal, it certainly is not good programming style. As mentioned above, the compiler will use ***implicit casts*** where it can deduce the correct conversion. If want to create a Class to handle multiple numeric types, do not construct it as an apparent Integer Class. This will lead to misunderstanding of your code.

Compare the results of the using an Integer Class with Double/Float arguments with the results using a Templated Class shown later in this section.

```
bool result4 = int_calc_obj.less_than(4.2, 4.5);   // bresult = FALSE
int result5   = int_calc_obj.add(3, 4.1);          // result = 7
int result6  = int_calc_obj.multiply(5.9, 10.9);   // result1 = 50

double result2  = int_calc_obj.add(3, 4.1);        // result2 = 7
double result3 = int_calc_obj.multiply(3.9, 4.2);  // result3 = 50
```

See ***Example-1a.cc*** for a working code example of **inc_calc_obj** usage, both good and bad.

## Converting the Integer Class into a Template Class

This Template Class definition is identical to the Integer Class definition when instantiated for an Integer value.

```
template <class RepType> class  int_calc
{
    public:
    bool less_than(RepType x, RepType y)     // Member Function defined in line
    {
        return(x < y);
    }
    RepType multiply(RepType x, RepType y)   // Member Function defined in-line
    {
        return x*y;
    }
    RepType add(RepType x, RepType y);       // Prototype only
};

// Member Function Body Defined Outside the Class
template <class RepType> RepType gen_calc<RepType>::add(RepType x, RepType y)
{
      return x+y;
};
```

***Notice*** that if you remove both instances of the text bounded in **red** (i.e. "**template <class RepType>**") and replace all other **Type** entries with "int", the Class definition is exactly the same as our original Integer Class.

### Instantiate an Integer version of the Template Class:

```
gen_calc<int>  int_calc_obj;
int result;

bool result1 = int_calc_obj.less_than(3, 8);      // True
int result2  = int_calc_obj.add(3, 4);            // result = 7
int result3  = int_calc_obj.multiply(5, 10);      // result = 50
```

### Instantiate a Double version of the Template Class:

```
gen_calc<double>  dbl_calc_obj;
double result;

bool result4 = dbl_calc_obj.less_than(3.1, 3.9);   // True
result5      = dbl_calc_obj.add(3.2, 4);           // result = 7.2
result6      = dbl_calc_obj.multiply(5.5, 10.2);   // result = 56.1
```

## Working Code Example (example-1a.cc)

```
#include <iostream>
using namespace std;

// Standard Class definition
class  int_calc
{
public:
    bool less_than(int x, int y)   // Member Function defined in line
    {
        return(x < y);
    }
    int multiply(int x, int y)   // Member defined in-line
    {
        return x*y;
    }
    int add(int x, int y);        // Member ProtoRepType Only
};

// Member Function Body Defined Outside the Class!
int int_calc::add(int x, int y)
{
    return   x + y;
};

inline const char * const BoolToString(bool b)
{
  return b ? "true" : "false";
}

int main()
{
  int_calc int_calc_obj;           // Instantiate an object of Class int_calc

  cout << "\nNormal Usage of <int_calc> Class\n";
  cout << "Test: less_than(3,4)   = " << BoolToString(int_calc_obj.less_than(3, 4)) << "\n";
  cout << "Test: add(3,4)         = " << int_calc_obj.add(3,4) << "\n";
  cout << "Test: multiply(5, 10)  = " << int_calc_obj.multiply(5,10) << "\n";

  cout << "\nAbusive Usage of <int_calc> Class\n";
  cout << "Test: less_than(4.2,4.5)   = " << BoolToString(int_calc_obj.less_than(4.2,4.5)) << "\n";
  cout << "Test: add(3, 4.1)          = " << int_calc_obj.add(3,4.1) << "\n";
  cout << "Test: Multiply(5.9,10.8)   = " << int_calc_obj.multiply(5.9,10.8) << "\n";

  double result1 = int_calc_obj.add(3.2,4.5);
  double result2 = int_calc_obj.multiply(3.2,4.4);
  cout << "\nTest: double result1 = add(3.2,4.4)     = " << result1 << "\n";
  cout << "Test: double result2 = multiply(3.2,4.4)  = " << result2 << "\n";

}
```

## Example-1a Output

```
$example-1a<cr>

Normal Usage of <int_calc> Class
Test: less_than(3,4)   = true
Test: add(3,4)         = 7
Test: multiply(5, 10)  = 50

Abusive Usage of <int_calc> Class
Test: less_than(4.2,4.5)   = false
Test: add(3, 4.1)          = 7
Test: Multiply(5.9,10.8)   = 50

Test: double result1 = add(3.2,4.4)       = 7
Test: double result2 = multiply(3.2,4.4)  = 12
```

Pay close attention to implicit ***casting*** demonstrated above, and the resulting output values. First, arguments are cast to Integer, (if they are Double or Float arguments).

## Working Code Example (example-1b.cc):

```
#include <iostream>
using namespace std;

template <class RepType> class gen_calc
{
public:
    bool less_than(RepType x, RepType y)       // Member Function defined in line
    {
        return(x < y);
    }
    RepType multiply(RepType x, RepType y)    // Member defined in-line
    {
        return x*y;
    }
    RepType add(RepType x, RepType);          // Member ProtoRepType Only
};

// Member Function Body Defined Outside the Class!
template <class RepType> RepType gen_calc<RepType>::add(RepType x, RepType y)
{
    return x + y;
};

inline const char * const BoolToString(bool b)
{
  return b ? "true" : "false";
}
```

```
int main()
{
  gen_calc<int> int_calc_obj;
  gen_calc<double> dbl_calc_obj;

  cout << "\nInteger Instantiation of gen_calc" << endl;
  cout << "Test: less_than(3,4)   = " << BoolToString(int_calc_obj.less_than(3, 4)) << endl;
  cout << "Test: add(3,4)         = " << int_calc_obj.add(3,4) << endl;
  cout << "Test: multiply(5, 10)  = " << int_calc_obj.multiply(5,10) << endl;

  cout << "\nDouble Instantiation of gen_calc" << endl;
  cout << "Test: less_than(4.2,4.5)   = " << BoolToString(dbl_calc_obj.less_than(4.2,4.5)) << endl;
  cout << "Test: add(3, 4.1)          = " << dbl_calc_obj.add(3,4.1) << endl;
  cout << "Test: Multiply(5.9,10.8)   = " << dbl_calc_obj.multiply(5.9,10.8) << endl;
}
```

## Example-1b Output:

```
$example-1b<cr>

Integer Instantiation of gen_calc
Test: less_than(3,4)   = true
Test: add(3,4)         = 7
Test: multiply(5, 10)  = 50

Double Instantiation of gen_calc
Test: less_than(4.2,4.5)   = true
Test: add(3, 4.1)          = 7.1
Test: Multiply(5.9,10.8)   = 63.72
```

Notice, that second to last ***add*** function also has an implicit cast, (from Integer to Double).

# Section 2: Converting a Function into a Template Function

In this example we show how to convert a simple add function into a Template function that can handle multiple data Types.

## Integer Add Function

```
int int_add( int x, int y)
{
  return x + y;
}
```

### Instantiate a Integer Function:

```
int result = int_add(1, 9);
```

## Template Add Function

As with the Template Class, converting a function to Template simply requires inserting the Template definition preamble shown in red (i.e. "**template <class RepType>")** and replacing the "int" word with Rep**Type.**

```
template <class  RepType>  RepType  gen_add (RepType  x,  RepType  y)
{
return x + y;
}
```

### Instantiate an Integer Version of the function

```
int result = gen_add<int>(1, 9);
```

### Instantiate a Double Version of the function

```
double result = gen_add<double>(1.1 , 9.4);
```

## Working Example Code (Example-2.cc):

As an experiment, the user should try to instantiate several new types of template classes and functions in the program below:

```
#include <iostream>
using namespace std;

// Normal function add(x,y)
int int_add(int x, int y)
{
    return x + y;
}

// Template add(x,y) Function
template <class RepType> RepType gen_add( RepType x, RepType y)
{
    return x + y;
}

int main()
{
  cout << "\nInteger Instantiation of int_add" << endl;
  cout << "Test: int_add(3,4)         = " << int_add(3,4) << endl;

  cout << "\nInteger Instantiation of gen_add" << endl;
  cout << "Test: gen_add(3,4)         = " << gen_add<int>(3,4) << endl;

  cout << "\nDouble Instantiation of gen_add" << endl;
  cout << "Test: gen_add(3.8, 4.1)    = " << gen_add<double>(3.8,4.1) << endl;
}
```

### Example-2 Output:

```
hydra(hp_docs}$example-2

Integer Instantiation of int_add
Test: int_add(3,4)         = 7

Integer Instantiation of gen_add
Test: gen_add(3,4)         = 7

Double Instantiation of gen_add
Test: gen_add(3.8, 4.1)    = 7.9
```

## Section 3: Template Type and Non-Type parameters

Template Parameters come in two flavors.

### Type Parameters

Type parameters are symbol-name substitutes for the real types specified when a template is instantiated. These include:

- Template arguments may be ordinary types such as ***Int*** and ***double***, and ***Classes***.
- Templates parameters may also be ***Templates*** (Classes, not functions).

### Non-Type Parameters

- A Template argument may be integral constant expressions or an enumeration.
  Real number and string literals are not valid constant expressions.
- A Template argument can be the address of an object or function that has external linkage.
  Pointers can be of the form ***&of***, where ***of*** is an object or function.
  Pointers can be of the form ***f*** where ***f*** is the name of a function.
- A Template argument can be a none-overloaded pointer to a member.
  Pointers to members must be in the form ***%X::of*** where ***X*** is a Class and ***of*** is the member.

## section 4: Template Functions containing Casts:

The nice thing about Templates is that they only require the minimal information to perform the correct function. If the compiler can determine unspecified Types from the information at hand, then explicit casts and Template Type specifications are not needed. For example, given the following multiply function Template, we may use the Template explicitly.

```
template <class m_RepType>  m_RepType multiply( m_RepType x, m_RepType y)
{
  Return( x, y );
}

int  myresult = multiply<int>(3, 5);
```

However, since the Type of myresult is known and the arguments are both Integers, the following also works.

```
int  myresult = multiply(3, 5);
```

Why you ask? Because the compiler knows enough to select the right multiply function canididate (i.e. **int multiply(int,int**) without explicitly using full Template Type specification (i.e. **multiply<int>)**. However, the following will fail unless there is a specific function candidate in the form **int multiply(int, double).**

```
int myresult = multiply(3, 2.1)
```

## Template Function Containing a Cast Expression

There are times when a user function needs more Type information to perform its assigned operation. The form of the ***cast expression*** is as follows:

```
unary-expression  ( RepType-name )  cast-expression
```

In the example below it is instantiated as: **return** (RepType) **x;**

Notice we have two different RepTypes specified in this Template. The second RepType is shown in Orange.

```
template <class RepType, class RepType2> RepType cast(RepType2 x)
{
  return (RepType)x;
}
```

## Instantiate a Double Cast Example:

```
double result;

result = cast<double>(10)
```

If you are confused, remember that Templates only specify everything necessary to allow the compiler to deduce the correct RepType. In this case "**cast<double>(10)**" will cast the Integer argument into a Double value

Note that arguments to be deduced *must always follow* arguments that must be specified. Thus, the parameter **RepType** is specified (i.e. **double**) and **RepType2** is deduced (i.e. **Integer**)!

## Instantiate an Integer Cast Example:

## Working Example Code (Example-3.cc):

```
int result;

result = cast<int>(10.4);
#include <iostream>

using namespace std;

//
// ============================================================================
//                            M A I N . C C
```

```
// ========================================================================
//

template <class RepType, class RepType2>  RepType genadd (RepType2 x, RepType2 y)
{
    return (RepType) (x + y);
}


int main()
{

    double res_dbl = genadd<double> (10, 2);
    int res_int    = genadd<int> (10.4, 3.6);

cout << "\nInterger add cast to Double   = " << res_dbl << "\n";
cout << "\nDouble add cast to Integer    = " << res_int << "\n";
```

## EXAMPLE-3 OUTPUT:

# Section 4: Specialization:

This document explores template specialization.

## Template Specialization

Normally, when working with templates, you write a generic version that can handle all allowable data Types. There may be times when some of the possible data Types may need special handling.

Well how do we do that? We create a secondary template specification that defines the special handling for a particluar Type.

The generic template specification is as follows:

```
template <class RepType> class mycontainer { ... };
```

To create a specialization for a particular Type, say <char>, we would specify it as follows:

```
template <> class mycontainer <char> { ... };
```

If you think about it, the specialization format makes sense. We have to tell the compliler that this is a template, however the parameters are already known because of the orginal template specification. The template <> simply indicates this a template specialization and the addition of <char> after the class specification, tells the compiler the <RepType> associated with that specialization.

## Class Template Framework

```
// class template:
template <class RepType>
class mycontainer {
    RepType element;
```

```
  public:
      mycontainer (RepType arg) { . . .}
 // Other Members ,,,
 };
```

Class Template Specialization Framework

```
 // class template specialization:
 template <>
 class mycontainer <char> {
      char element;
  public:
      mycontainer (char arg) { . . . }
 // Other Members ,,,
 };
```

## Working Example Code (Example-4.cc):

```
// template specialization
#include <iostream>

using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element >= 'a')&&(element <= 'z'))
      element+='A'-'a';
```

```
      return element;
    }
};

int main () {
  mycontainer <int> myint (7);
  mycontainer <char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

### Example-4 Output:

## Section 5: Template Functions containing Casts:

This document explores using a Class Templates that themselves contain Templated Functions and *the possible problems this may cause the programmer*.

# Template Class containing a Template

It is possible for a Templated Class to have a member function that is itself a Template. For Example:

```
template<class RepType>
   class C
   {
   public:
      template<class RepType2>
        RepType2 func(RepType2 y);         {
           cout << "\nY= " << y << "\n\n";
           return y;
        }
};
```

In the above case, things are rather straight forward. But when the class just defines the member function interfaces, programmers often get in trouble. In the following example, just the interface is defined:

```
template<class RepType>
```

```
   class C
   {
   public:
      template<class RepType2>
        RepType2 func(RepType2 y);
   }
};
```

## Defining Template Function outside the Templated Class Specification

The inexperienced programmer will often try define the function as follows:

```
// BAD CODE
template <class RepType, class RepType2> RepType2 C<RepType>::func(RepType2 arg)
{
// .. some code...
}
```

The proper way to define a Templated Function outside the Class Template

When defining a Templated Function outside of a Class Template, you must respect this by using the template keyword twice. Once for the Class and once for the Function:

```
template <class RepType>          // For the class
   template <class RepType2>     // For the function
      RepType2 C<RepType>::func(RepType2 arg)
      {
            // code
      }
```

### Instantiate the class C and call func:

```
       char a = 'a';
       char b;

      C<int> c;           // Instantiate Class 'C'
       b = c.func(a);    // Call 'func'

One last thing, the call to 'func' could also be written as:

       b = c.func<char>(a);    // Call 'func'
```

But as stated previously, Templates only 'need' to specify everything necessary to allow the compiler to deduce the correct Type. Just follow the rule:

Arguments to be *deduced* must always follow the arguments that must be *specified*..

## Working Example Code (example-5.cc):

```
#include <iostream>

using namespace std;

template<class X>
  class C
  {
  public:
    template<class Y> void f(Y);
  };

template<class X>
  template<class Y>
    void C<X>::f(Y y)
    {
      cout << "\nY = " << y << "\n\n";
    }

int main()
  {
  C<int> c;
  char a = 'a';
  c.f(a);
  return 0;
  }
```

## Example-5 Output:

# Company Information

## Hawthorne-Press.com

**10310 Moorberry Lane**

**Houston, Texas 77043, USA**

**Phone:** **+1 (832) 630-6822**

**Hawthorne Press Website**: www.hawthorne-press.com

# Revision History

| Date | By | Section | Changes |
|---|---|---|---|
| **10/04/2014** | C.E. Thornton | All | Initial Document |
| **05/27/2018** | C.E. Thornton | Copyright | Update Address |
| | | | |