C++ Templates Explained In Color

REVISION 0

Hawthorne-Press.com

C++ Templates Explained In Color

Published by

Hawthorne-Press.com

10310 Moorberry Lane
Houston, Texas 77043, 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 six sections. **At this time, only two sections have been written.**

1. Converting a Normal Class into a Template Class
2. Converting a Function into a Template Function
3. *Template parameters and type specifications*
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 when document is completed.

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. *Hint*, Double and Float arguments are cast to Integer before any other operations are performed. Compare the results with similar results in **example-1b.cc.**

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). However, in this case it does not change the result because it's cast to the ***proper type*** for this specialization of the template.

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 RepType.

```
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
```

Sections 3 – 6 in progress!

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