TEMPLATE IMPLEMENTATION & COMPILER (.H OR .CPP?) - 2020

Sponsor Open Source development activities and free contents for everyone.

Donate with PayPal

Thank you.

- K Hong (http://bogotobogo.com/about_us.php)

Should implementation be included in .h file?

In this section, we'll discuss the issue of template implemention (definition) and declaration (interface), especially related to where we should put them: header file or cpp file?

Quite a few questions have been asked like these, addressing the same fact:

1. Why can templates only be implemented in the header file?
2. Why should the implementation and the declaration of a class template be in the same header file?
3. Why do class template functions have to be declared in the same translation unit?

4. Do class template member function implementations always have to go in the header file in C++?

For other template topics, please visit
http://www.bogotobogo.com/cplusplus/templates.php
(http://www.bogotobogo.com/cplusplus/templates.php).

As we already know, template specialization (or instantiation) is another type of polymorphism where choice of function is determined by the compiler at compile time, whereas for the virtual function, it is not determined until the run time.

Template provides us two advantages:
First, it gives us type flexibility by using generic programming.
Second, its performance is near optimal.
Actually, a compiler does its best to achieve those two things.
However, it also has some implications as well.

Stroustrup explains the issue in his book "Programming Principles and Practice Using C++:

> As usual, the benefits have corresponding weaknesses. For templates, the main problem is that the flexibility and performance come at the cost of poor separation between the "inside" of a template (its definition) and its interface (its declaration).
> When compiling a use of a template, the compiler "looks into" the template and also into the template argument types. It does do to get the information to generate optimal code. To have all the information available, current compilers tend to require that a template must be fully defined whenever it is used. That includes all of its member functions and all template functions called from those. Consequently, template writers tend to place template definition in header files. That is not actually required by the standard, but until improved implementations are widely available, we recommend that you do so for your own templates: **place the definition of any template that is to be used in more than one translation unit in a header file**.

When we do:

```
template<typename T>
class Queue
{
...
}
```

it's important for us to realize that the template is not class definition yet. It's a set of instructions to the compiler about how to generate the class definition. A particular realization of a template is called an **instantiation** or a **specialization**.

Unless we have a compiler that has implemented the new **export** keyword, placing the template member functions in a separate implementation file won't work. Because the templates are not functions, they can't be compiled separately.

Templates should be used in conjunction with requests for particular instantiations of templates. So, the simplest way to make this work is to place all the template information in a header file and to include the header file in the file that the template will be used.

Sample code - recommended approach

The code has generic implementation of queue class (**Queue**) with simple operations such as **push()** and **pop()**.

The **foo()** does **int** specialization, and **bar()** does **string**. The declaration and definition are all in one header file, **template.h**. Each of the **foo.cpp** and **bar.cpp** includes the same **template.h** so that they can see both of the declaration and definition:

template.h

Programming II - gcc ARM Toolchain and Simple Code on Ubuntu and Fedora (/cplusplus/embeddedSystemsF

Embedded Systems Programming III - Eclipse CDT Plugin for gcc ARM Toolchain (/cplusplus/embeddedSystemsF

Exceptions (/cplusplus/exceptions.php)

Friend Functions and Friend Classes (/cplusplus/friendclass.php)

fstream: input & output (/cplusplus/fstream_input_outpu

Function Overloading (/cplusplus/function_overloadin

Functors (Function Objects) I - Introduction (/cplusplus/functor_function_ob

Functors (Function Objects) II - Converting function to functor (/cplusplus/functor_function_ob

Functors (Function Objects) - General (/cplusplus/functors.php)

Git and GitHub Express... (/cplusplus/Git/Git_GitHub_Expr

GTest (Google Unit Test) with Visual Studio 2012 (/cplusplus/google_unit_test_gte

Inheritance & Virtual Inheritance (multiple inheritance) (/cplusplus/multipleinheritance.

Libraries - Static, Shared (Dynamic) (/cplusplus/libraries.php)

Linked List Basics (/cplusplus/linked_list_basics.ph

Linked List Examples (/cplusplus/linkedlist.php)

```
#include <iostream>

// Template Declaration
template<typename T>
class Queue
{
public:
    Queue();
    ~Queue();
    void push(T e);
    T pop();
private:
    struct node
    {
        T data;
        node* next;
    };
    typedef node NODE;
    NODE* mHead;
};

// template definition
template<typename T>
Queue<T>::Queue()
{
    mHead = NULL;
}

template<typename T>
Queue<T>::~Queue()
{
    NODE *tmp;
    while(mHead) {
        tmp = mHead;
        mHead = mHead->next;
        delete tmp;
    }
}

template<typename T>
void Queue<T>::push(T e)
{
    NODE *ptr = new node;
    ptr->data = e;
    ptr->next = NULL;
    if(mHead == NULL) {
        mHead = ptr;
        return;
    }
    NODE *cur = mHead;
    while(cur) {
        if(cur->next == NULL) {
            cur->next = ptr;
            return;
        }
        cur = cur->next;
    }
}

template<typename T>
T Queue<T>::pop()
{
    if(mHead == NULL) return NULL;
    NODE *tmp = mHead;
    T d = mHead->data;
    mHead = mHead->next;
    delete tmp;
```

make & CMake (/cplusplus/make.php)

make (gnu) (/cplusplus/gnumake.php)

Memory Allocation (/cplusplus/memoryallocation.p

Multi-Threaded Programming - Terminology - Semaphore, Mutex, Priority Inversion etc. (/cplusplus/multithreaded.php)

Multi-Threaded Programming II - Native Thread for Win32 (A) (/cplusplus/multithreading_win3

Multi-Threaded Programming II - Native Thread for Win32 (B) (/cplusplus/multithreading_win3

Multi-Threaded Programming II - Native Thread for Win32 (C) (/cplusplus/multithreading_win3

Multi-Threaded Programming II - C++ Thread for Win32 (/cplusplus/multithreading_win3

Multi-Threaded Programming III - C/C++ Class Thread for Pthreads (/cplusplus/multithreading_pthr

MultiThreading/Parallel Programming - IPC (/cplusplus/multithreading_ipc.p

Multi-Threaded Programming with C++11 Part A (start, join(), detach(), and ownership) (/cplusplus/multithreaded4_cplu

Multi-Threaded Programming with C++11 Part B (Sharing Data - mutex, and race conditions, and deadlock) (/cplusplus/multithreaded4_cplu

Multithread Debugging (/cplusplus/multithreadedDebu

Object Returning (/cplusplus/object_returning.php

Object Slicing and Virtual

```
    return d;
}
```

foo.cpp

```
#include "template.h"
void foo()
{
    Queue<int> *i = new Queue<int>();
    i->push(10);
    i->push(20);
    i->pop();
    i->pop();
    delete i;
}
```

bar.cpp

```
#include "template.h"
void foo()
{
    Queue<std::string> *s = new Queue<std::string>();
    s->push(10);
    s->push(20);
    s->pop();
    s->pop();
    delete s;
}
```

Sample code 2 - breaking up header file

We could breakup the header into two parts: declaration (interface) and definition (implementation) so that we can keep the consistency regarding the separation of interface from implementation. We usually name the interface file as **.h** and name the implementation file as **.hpp**. However, the end result is the same: we should include those in the **.cpp** file.

Class Template vs Template Class

There is a delicate but significant distinction between **class template** and **template class**:

1. Class template is a template used to generate template classes.
2. Template class is an instance of a class template.

Table (/cplusplus/slicing.php)

OpenCV with C++ (/cplusplus/opencv.php)

Operator Overloading I (/cplusplus/operatoroverloading

Operator Overloading II - self assignment (/cplusplus/operator_oveloading

Pass by Value vs. Pass by Reference (/cplusplus/valuevsreference.ph

Pointers (/cplusplus/pointers.php)

Pointers II - void pointers & arrays (/cplusplus/pointers2_voidpoint

Pointers III - pointer to function & multi-dimensional arrays (/cplusplus/pointers3_function_

Preprocessor - Macro (/cplusplus/preprocessor_macro

Private Inheritance (/cplusplus/private_inheritance.

Python & C++ with SIP (/python/python_cpp_sip.php)

(Pseudo)-random numbers in C++ (/cplusplus/RandomNumbers.pl

References for Built-in Types (/cplusplus/references.php)

Socket - Server & Client (/cplusplus/sockets_server_clien

Socket - Server & Client 2 (/cplusplus/sockets_server_clien

Socket - Server & Client 3 (/cplusplus/sockets_server_clien

Socket - Server & Client with Qt (Asynchronous / Multithreading / ThreadPool etc.) (/cplusplus/sockets_server_clien