C++ Template Classes and Friend Function Details

Wednesday, April 28, 2021 8:55 PM

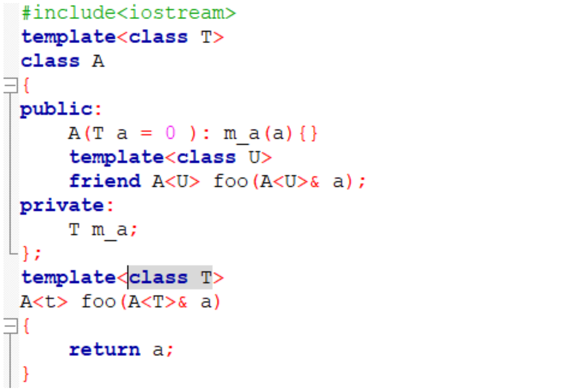
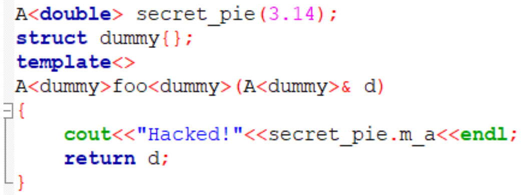
C++ Template classes and Friend Function Details

Introduction

- There are some situations that we want to create a friend function such as implementing operator<<
- There are straightforward to write for non-template classes, but templates are useful in this situation.
- Each way of setting up template classes with friend functions behaves different way.

Some terminology

- **Template instantiation :** specific instance of templated item
  + Example: templated class A, A<int> is a specific instantiation of A
- **Non-member function:** a function that is not a member of a class:
  + Example:
  Approach #1:

<table>
<tr>
<td>
<pre>
#include<iostream>
template<class T>
class A
{
public:
    A(T a = 0): m_a(a){}
    template<class U>
    friend A<U> foo(A<U>& a);
private:
    T m_a;
};
template<class T>
A<t> foo(A<T>& a)
{
    return a;
}
</pre></td>
<td>- declaring a templated friend function of a templated class.
- works for all cases where we want to explicitly pass an instance of class A to foo()</td>
<td>Problems:
All template instaniations of A are friends with all template instaniations of foo()
- Example:
A<int> is friends with foo<int> but also foo<double&gt;</td>
</tr>
<tr>
<td>
<pre>
A<double> secret_pie(3.14);
struct dummy{};
template<>
A<dummy>foo<dummy>(A<dummy>& d)
{
    cout<<"Hacked!"<<secret_pie.m_a<<endl;
    return d;
}
</pre></td>
<td>The line cout should not work</td>
<td>To prevent this potential error, we must take a different approach</td>
</tr>
</table>

#Approach 2

<table>
<tr>
<td><pre>
template<class T>
class A;

template<class T>
A<T> foo(A<T>& a);

template<class T>
class A
{
  public:
    A(T a = 0): m_a(a) {}

    friend A foo<T>(A& a);

  private:
    T m_a;
};

template<class T>
A<T> foo(A<T>& a)
{
  return a;
}
</pre></td>
<td>- foo() is declared as a template function using a declared templated class A
- When defining A, we make each template instantiation of A friends with the corresponding template instantiation of foo()
- In this case A<int> is friends with foo<int> but foo<double> is not a friend of A<int> since the type in their template parameters do not match
- In certain cases, when nummeric type objects, the friend funtions have side effect: the parameters passed to them must be explicitly associated with the expected parameter types for template arguement</td>
</tr>
</table>

- I spend more than 3 hours trying to read the article but it doesn't help so I decide to note some basic things about Template class, Template function, Friend function, Late bindings and early bindings

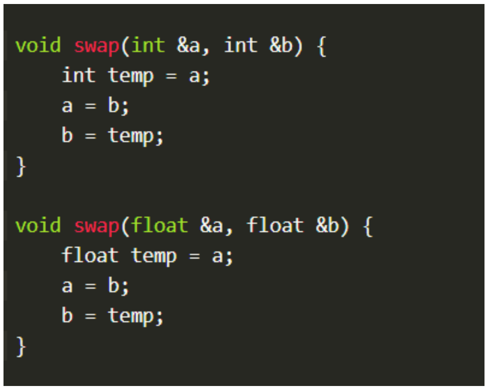
Function and class template

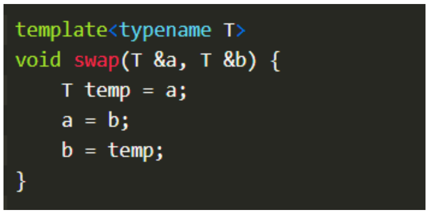
- template is a keyword in C++, it represents abtract types of data like int, float, class,.....

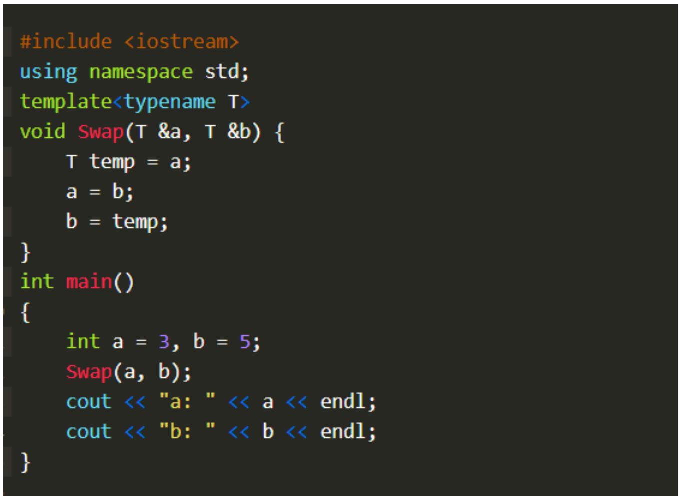
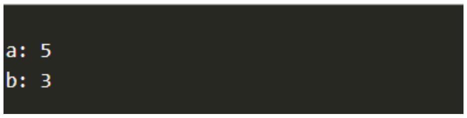
? Why would we use template?

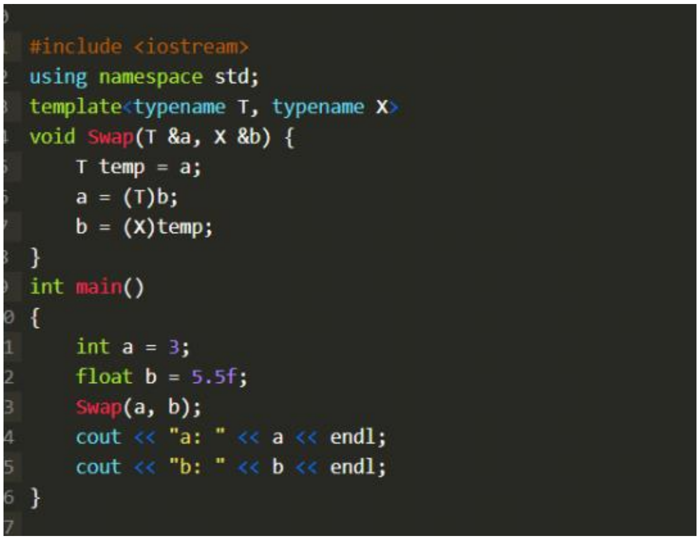
- Instead of writing overloading function for every methods for every types, we just need to write one template for all

Function template

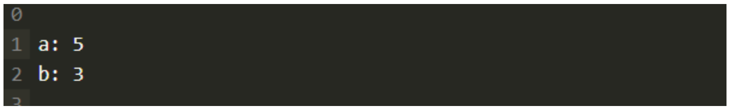

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
 	
void swap(float &a, float &b) {
    float temp = a;
    a = b;
    b = temp;
}
	these are two the same function but different in the functions' types


template<typename T>
void swap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}
	instead of writing 2 functions, we just need 1 template function


#include <iostream>
using namespace std;
template<typename T>
void Swap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}
int main()
{
    int a = 3, b = 5;
    Swap(a, b);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
}
	
	a: 5
	b: 3



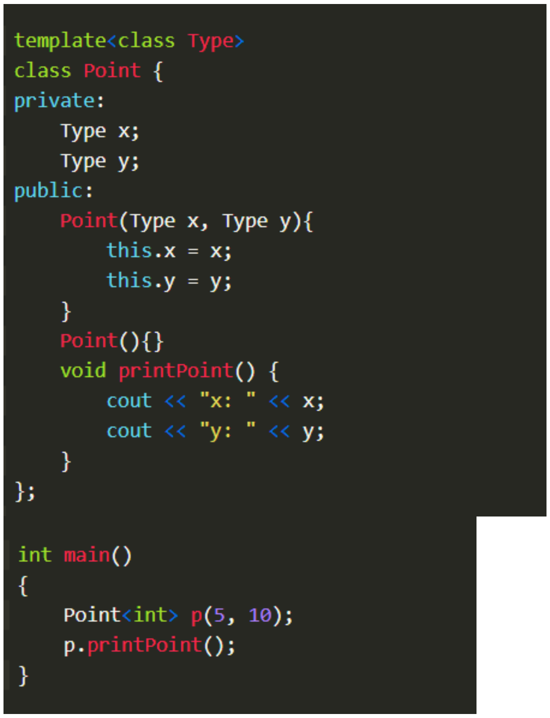
```cpp
#include <iostream>
using namespace std;
template<typename T, typename X>
void Swap(T &a, X &b) {
    T temp = a;
    a = (T)b;
    b = (X)temp;
}
int main()
{
    int a = 3;
    float b = 5.5f;
    Swap(a, b);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
}
```



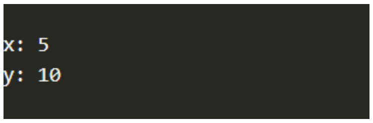
```
a: 5
b: 3
```

Class template

Example: creating a class called Point containing x,y with whatever type



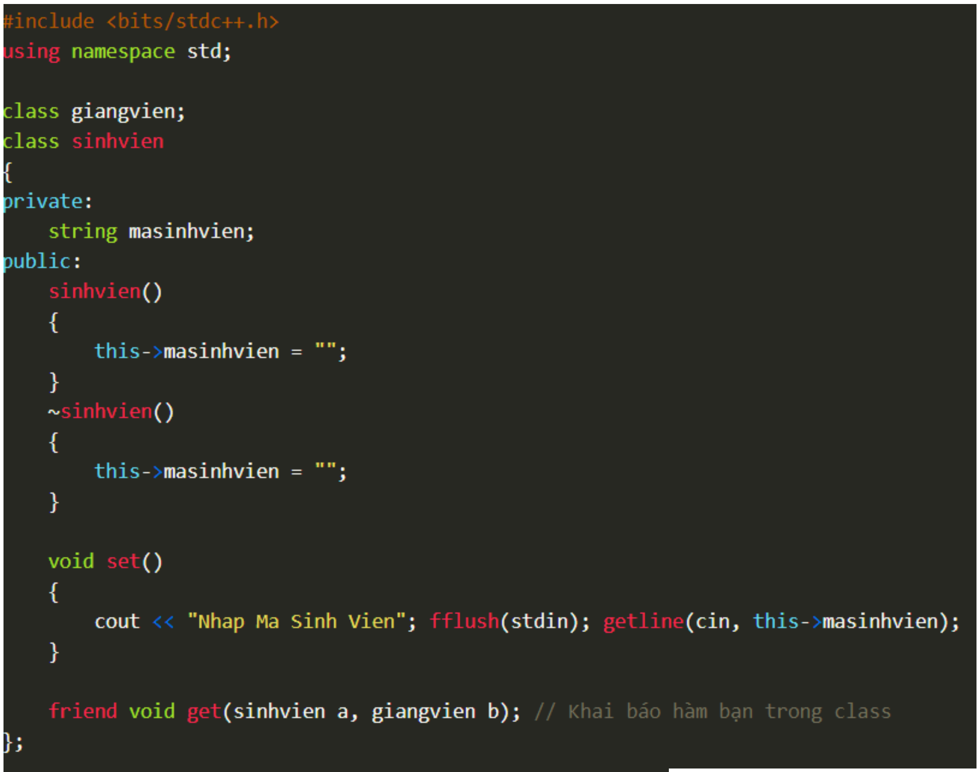
```cpp
template<class Type>
class Point {
private:
    Type x;
    Type y;
public:
    Point(Type x, Type y){
        this.x = x;
        this.y = y;
    }
    Point(){}
    void printPoint() {
        cout << "x: " << x;
        cout << "y: " << y;
    }
};

int main()
{
    Point<int> p(5, 10);
    p.printPoint();
}
```



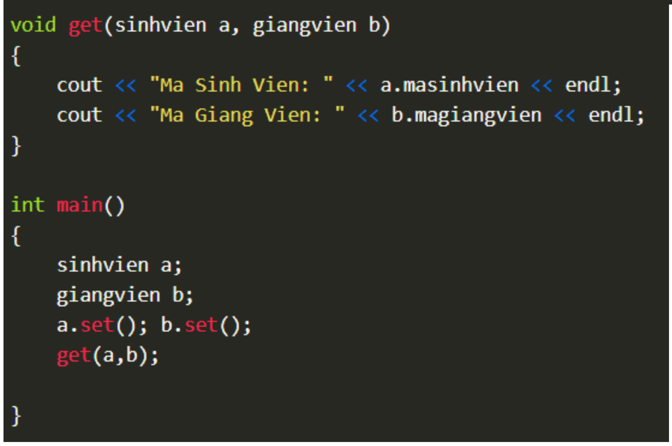
```
x: 5
y: 10
```

Friend function:

- Is a free function, not belong to any class but this function can access to every member of the class including private members



```
#include <bits/stdc++.h>
using namespace std;

class giangvien;
class sinhvien
{
private:
    string masinhvien;
public:
    sinhvien()
    {
        this->masinhvien = "";
    }
    ~sinhvien()
    {
        this->masinhvien = "";
    }

    void set()
    {
        cout << "Nhap Ma Sinh Vien"; fflush(stdin); getline(cin, this->masinhvien);
    }

    friend void get(sinhvien a, giangvien b); // Khai báo hàm bạn trong class
};
```

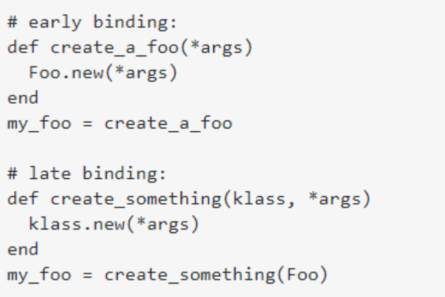


```
void get(sinhvien a, giangvien b)
{
    cout << "Ma Sinh Vien: " << a.masinhvien << endl;
    cout << "Ma Giang Vien: " << b.magiangvien << endl;
}

int main()
{
    sinhvien a;
    giangvien b;
    a.set(); b.set();
    get(a,b);

}
```

Early binding and late binding

- Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.)
  ++++Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.
- The opposite of early binding is late binding. Late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time.
- The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times. However today, fast computers have significantly reduced the execution times related to late binding.

```
# early binding:
def create_a_foo(*args)
  Foo.new(*args)
end
```

- 

```
# early binding:
def create_a_foo(*args)
  Foo.new(*args)
end
my_foo = create_a_foo

# late binding:
def create_something(klass, *args)
  klass.new(*args)
end
my_foo = create_something(Foo)
```