C++ PROGRAMMING

DR. YIHSIANG LIOW (JULY 23, 2025)

Contents

72. Class Templates

OBJECTIVES

- Write class templates
- Instantiate classes from class templates

The problem

What if we want to have a class where the only instance variable is an integer, and then we want another where the only instance variable is a character?

They both have the same kind of "behavior". The constructor sets the value of the instance variable and there are get and set methods. Here's an example. Make sure you run it.

```
class Int
{
public:
     Int(int x0)
         : x(x0)
     {}
     int get() const { return x; }
     void set(int x0) { x_ = x0; }
private:
     int x;
};

class Char
{
public:
     Char(char x0): x(x0) {}
     char get() const { return x; }
     void set(char x0) { x = x0; }
private:
     char x;

};
```

The test code is:

```
int main()
{
    Int d(5);
    Char e(' a');
    std::cout << d.get() << ' '
              << e.get() << '\n';
    return 0;
}
```

Note that the two classes are very similar. In fact, within the class the only difference is the type associated with the instance variable.

```
class Int
{
public:
    Int(int x0)
        : x(x0)
    {}
    int get() const { return x; }
    void set(int x0) { x_ = x0; }
private:
    int x;
};

class Char
{
public:
    Char(char x0): x(x0) {}
    char get() const { return x; }
    void set(\EMPHASIZE{char} x0) { x = x0; }
private:
    char x;

};
```

What a pain to type two chunks of code which are almost the same. It's like typing unrolled for-loops!

Class templates

Now we allow types to vary.

Generic Programming refers to the style of programming using parameters for types.

```
template < typename T >
class D {
public:
     D(){}
     D(T c)  : x(c)  {}
     T get() const { return x; }
     void set(const T & c) { x = c; }
private:
     T x;
};

int main()
{
    D <int> d(5);
    D <char> e('a');
    std::cout << d.get() << ","
              << e.get() << "\n";
    return 0;
}
```

D is called a class template.
T is a type argument

Class template D with T= int.
D<int>is a class.

D<char>class

A class is generated from a class template by the compiler when you specify the parameters.

Example:

`D < int > d(5);`

If `D < char >` does not appear, then the class `D < char >` is not generated by the compiler. Here are some terms:

- **Template instantiation** = generating class from class template and template arguments.
- **Specialization** = template with parameters specified
- **Template members** = members of a class template.

Template members are parameterized by the templates. So if a template member is defined outside the class, then the class template must be specified

WARNING: Everything must be placed in the header file!!! No .cpp for class template!!!

```
#include <iostream>

template< typename T >
class D {
public:
     D(){}
     D(T c) : x(c) {}
     T get() const;
     void set(const T &);
private:
     T x;
};

template< class T >
T D < T >::get() const
{
     return x;
}

template< typename T>
void D < T >::set(const T & c)
{
     x = c;
}

int main()
{
     D < int > d(5);
     D < char > e(' a');
     std::cout << d.get() << ","
               << e.get() << "\n";
     return 0;
}
```

Member function parameterized by `T`

WATCH OUT! Not `D` but `D< T >`

Note that

```
template< typename T >

T D < T >::get < T >() const
{
     return x;
}
```

is the same as:

```
template< typename T>

T D < T >::get() const
{
     return x;
}
```

i.e., within the scope of `D < T >`, T is already known.

Constants for template parameters

There are three things you can specify as template parameters. We'll talk about only two. The template parameters can be for

- types

or

- basic type values

In the above examples, the template parameters are for types:

```
template  < typename T >
class X
{
...
private:
     T ...
...
};
```

Here's an example where the template parameter is a value. Run this and study it very carefully.

```
#include < iostream >

template < typename T, int MAX >
class D
{
public:
     T get(int) const;
     void set(int, const T &);
private:
     T x[MAX];
};

template < typename T, int MAX>
T D < T, MAX >::get(int index) const
{
     return x[index];
}

template < typename T, int MAX>
void D < T, MAX >::set(int index, const T & c)
{
     x[index] = c;
};

int main()
{
     D < int, 10 > d; // d is an int array of size 10
     D< char, 5 > e; // d is a char array of size 5
     d.set(1,5);
     e.set(2,' a');
     std::cout << d.get(1) << ","
               << e.get(2) << "\n";
     return 0;
}
```

D is a class for fixed size arrays of different types. One of the template parameter is for the type of the array and the other template parameter is for the size of the array.

When instantiating the value must be a **constant expression.**

Run this:

```
const int x = 4;
int y = 5;
D< int, 5> d1;
D< int, x - 2> d2;
D< int, y> d3; //WRONG!!!  y is a
   variable.
```

Type parameters are:

- Any type including basic types (int, bool) or classes.

If D is a class template

```
template < typename T1, typename T2,
           int x, bool b >
class D{...}
```

Then

```
D < X, Y, 10, true >
```

and

```
const int x = 12;
D < X, Y, x - 2, !false >
```

are considered the same type (if `X` and `Y` are valid types).

Recall that typedef is only an alias and does not create a new type. So if

```
template< class X >
class D{...};
typedef int Salary;

int main()
{
     D < int > x;
     D < Salary > y;
}
```

then, `D < int >`, `D < Salary >` are of the same type and `x`, `y` have the same type.

Error checking can be done with two different levels: syntax errors in template and when the template parameters are being used.

In the definition of the template and class template, the template parameters are not specified yet. So the definition of

`Array < T >::print` is OK
`Array < int >::print` is OK

```
class vec2d
{
public:
     vec2d(float x0, float y0)
         : x(x0),y(y0)
     {}
private:
     float x, y;
};

template < typename T >
```

```
class Array {
public:
     Array(int s)
         : size(s), arr(new T[s])
     {}
     void print() const;
private:
     int size; T * arr;
};

template < typename T >
void Array< T>::print()
{
     for (int i=0; i< size; i++)
     std::cout << arr[i] << '\n';
}

int main()
{
     Array< int> a(10); a.print();
     Array < vec2d > b(10);
     b.print();
     return 0;
}
```

What's wrong?

You can specify default values for template parameters.

The rules for default parameters are like those for default values for function parameters.

```
#include <iostream>

class C{};

template < typename T0, typename T1=int >

class D{
public: void f(T1 a) { std::cout << a << "\n"; }
};

int main()
{
     D < C > obj1; obj1.f(1.1);
     D < C, double > obj2; obj2.f(1.1);
     return 0;
}
```

Default values are on the right

```
#include <iostream>

class C{};
```

```
template < typename T0, typename T1=int>
class D
{
public: void f(T1);
};

template < typename T0, typename T1 >
void D< T0, T1>::f(T1 a)
{
     std::cout << a << ``\n'';
}

int main()
{
     D< C> obj1;
     obj1.f(1.1);
     D< C, double> obj2;
     obj2.f(1.1);
}
```

Defining `f` outside the class. Like default values for function arguments, do not specify default value here.

When programming a class template, try it out for a specific type first. After the code works, change to a generic type and include template.

WARNING: The following will give you an error.

```
template < typename T > class C{...};
template < typename T > class D{...};

int main()
{
     C< D< int>> c;
     return 0;
}
```

Why? Because >> is an operator and for this code:

```
     C< D< int>> c;
```

you compiler will think you are trying ot do this:

```
     ...int >> c...
```

as if you're doing some input!!!

So write

```
C < D < int > > c;
```

to prevent the above. In general when working with templates, always have a space on the right of <and a space to the left of >.

Function and struct templates

So far we' ve talked about class templates which are templates for creating classes. But you can also have function templates and struct templates. Struct templates are very similar to class templates. Here's an example of a function template:

Run this and study it carefully:

```
#include <iostream>

template  < typename T >
T max(T x, T y)
{
     return (x >= y ? x : y);
}

int main()
{
     std::cout << max < int >(5, 2) << std::endl;
     std::cout << max < double >(1.2, 3.4) << std::endl;
     return 0;
}
```

For the case of function templates, usually the compiler can figure out what types are intended for the type parameter. So you can run the above test cases with this:

```
...

int main()
{
     std::cout << max(5, 2) << std::endl;
     std::cout << max(1.2, 3.4) << std::endl;
     return 0;
}
```

i.e., without the `< int >` and the `< double >`.

Like class templates, if you want to write a function template in a separate file, then the whole function template must be a header file.

Exercise -1.0.1. Write a min function template. Test it.

Here's another example for you to study. Add test code in your `main()` function and run it. Study it carefully. Note that in this case, the type parameter `T` appears in the body of the function template.

```
template  < typename T >

T swap(T & x, T & y)
{
     T t = x;
     x = y;
     y = t;
}
```

Exercise -1.0.2. Write a print function template to print an array of type `T` values. The function should receive an array of `T` values and the number of things in the array to print starting at index 0:

```
template  < typename T >

void print(T x[], int n)
{
     ...
}
```

Exercise -1.0.3. Write a linearsearch function template:

```
template  < typename T >

int linearsearch(T x[], int n, const T & target)
{
     ...
}
```

Test it! Make sure you test it with an array of integer values, double values, and char values.

Exercise -1.0.4. Write a bubblesort function template.

```
template  < typename T >
void bubblesort(T a[], int size);
```

Test it by performing bubble sort on an array of integers and an array of doubles. Is it possible to write this version?

```
template  < typename T >
void bubblesort(T * start, T * end);
```

Exercise -1.0.5. Write a binarysearch function template:

```
template  < typename T >
int binarysearch(T a[], int size, const T & target);
```

Test it! What about this:

```
template  < typename T >
T * binarysearch(T * start, T * end, T * target);
```

(I have a set of notes just on function templates and just on struct templates. Study them both when you have time.)

Typename and class

Instead of

```
template  < typename X >
class C{};
```

You can say

```
template  < class X >
class C{};
```

They mean the same thing. However, `typename` is better.

Template specialization

Suppose you have this **`Array`** class template:

```
#include <iostream>

template < typename T, int size >
class Array
{
public:
     Array(T x[])
     {
          for(int i = 0; i  < size; ++i)
          x_[i] = x[i];
     }
     int sum() const
     {
          int s = 0;
          for (int i = 0; i  < size; ++i)
          s += x[i];
          return s;
     }
private:
     T x_[size];
};

int main()
{
     int x[] = {2, 3, 5, 7, 11, 13, 17, 19};
     Array < int, 4 > a(x);
     std::cout << a.sum() << '\n';
     Array < int, 3 > b(x);
     std::cout << b.sum() << '\n';
}
```

Run it.

Then you realize that in your software that you are building, one extremely common usage of your **`Array`** library is for the case of modeling **`int` arrays of size 3**.

A loop over an array of size 3 to compute the sum is definitely slower that unrolling the loop. The unrolled loop is going to be about twice as fast.

Hmmm . . . can we **speedup this case?**

Yes you can . . . you can write a version of **`Array`** just for the case when T = int and size = 3. This is called a template specialization.

Run this:

```
#include <iostream>

template < typename T, int size >
class Array
{
     // ... same as before ...
};

template<>
class Array < int, 3 >
{
public:
     Array(int x[])
     {
          std::cout << "specialization speedup!!!\n";
          x_ = x[0]; y_ = x[1]; z_ = x[2];
     }
     int sum() const
     {
          std::cout << "specialization speedup!!!\n";
          return x_ + y_ + z_;
     }
private:
     int x_, y_, z_;
};

int main()
{
     // ... same as before ...
}
```

Template specialization is a very important idea and is used quite frequently.

Exercise -1.0.6. Add **`operator[]`** to both classes. Test your code.

Exercise -1.0.7. Then you realize that . . . wait a minute . . . this specialization idea is also helpful for a **`double`** array of size 3 and a **`float`** array of size 3. Hmmm . . . can we partially specialize? In other words can we have **`size`** = 3 but retain **`T`** as a template parameter Try to figure it out **`on your own`** before turning the page for the answer!!! You have 20 minutes ...

Exercise -1.0.8. Write a **`Matrix`** class template. A matrix is just a 2D array. Here's a sample run:

```
#include <iostream>
#include "Matrix.h"

int main()
{
     int x[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
     Matrix < int, 2, 3 > m(x); // m is 2-by-3.
     std::cout << m << '\n' // prints
                            // 0 1 2
                            // 3 4 5
     Matrix < int, 2, 1 > n(x); // n is 2-by-1
     std::cout << n << '\n' // prints
                            // 0
                            // 1
    return 0;
}
```

That 2-by-1 matrix is sometimes called a "2-dimensional column vector". Column vectors are used frequently. Furthermore processing a regular 1-dimensional array of size 2 is faster than viewing it as a 2-by-1 2D array. So . . . speedup your **`Matrix`** library by providing a specialization for the column vector case of a **`Matrix`** object where you use a 1-dimensional array in this case The following should then work:

```
#include <iostream>
#include "Matrix.h"

int main()
{
     int x[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
     Matrix < int, 2, 3 > m(x); // m is 2-by-3.
     std::cout << m << '\n' // prints
                            // 0 1 2
                            // 3 4 5
     Matrix < int, 2 > n(x); // n is 2-by-1
     std::cout << n << '\n' // prints
                            // 0
                            // 1
     return 0;
}
```

Such a template library is very common in computer graphics and computer vision. Such a library usually provide a typedef for **`Matrix < double, 4, 4 >`**, **`Matrix < int, 2 >`**, etc.. For instance there might be a typedef **`mat4x4`** for **`Matrix < double, 4, 4 >`**, **`matf4x4`** for **`Matrix < float, 4, 4 >`**, and **`vec4f`** for **`Matrix < float, 4 >`**.

WARNING . . . INCOMING SPOILER . . .

Here's the (obvious?) answer to the question on the previous page:

```
template < typename T, int size >
class Array
{
public:
     Array(T x[])
     {
          for(int i = 0; i  < size; ++i)
          x_[i] = x[i];
     }
     int sum() const
     {
          int s = 0;
          for (int i = 0; i  < size; ++i)
              s += x_[i];
          return s;
     }
private:
     T x_[size];
};

template < typename T >
class Array < T, 3 >
{
public:
     Array(T x[])
     {
          std::cout << "specialization speedup!!!\n";
          x_ = x[0]; y_ = x[1]; z_ = x[2];
     }
     int sum() const
     {
          std::cout << "specialization speedup!!!\n";
          return x_ + y_ + z_;
     }
private:
     T x_, y_, z_;
};
```