COMP 3522

Object Oriented Programming in C++

Week 10 day 1

Agenda

1. Template programming AKA C++ generics

COMP 3522

TEMPLATE PROGRAMMING

nonTemplate.cpp

STOP! WHAT'S A TEMPLATE?

- We're talking about the Standard **Template** Library
- What's a template?
- It's like a Java generic.

- It's that easy.

The C++ template is like Java’s generic

Our function is prefaced with either:

```
template <typename T>
```

or:

```
template <class T>
```

These are equivalent **template parameters**.

This works very nicely with functions!

```
template <typename T>
T get_max(T a, T b)
{
    return (a > b ? a : b);
}

// Suppose we have two doubles first and second
double maximum = get_max<double>(first, second);
```

But what actually happens?

1. The compiler uses the template to generate a new function
2. Each template parameter is replaced with the type passed as the actual template parameter
3. The function is called.

- This is automatic and invisible
- The compiler will often be able to determine the correct instantiation

But what actually happens?

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

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

https://www.geeksforgeeks.org/templates-cpp/

What about types?

Will this compile? What is the output?

```
int first{1};
double second{3.14};
double maximum = get_max(first, second);
// auto maximum = get_max(first, second);
```

templateFun.cpp

How about this?

```
int first{1};
double second{3.14};
double maximum = get_max<double>(first, second);
```

What about this?

```
int first{1};
double second{3.14};
int maximum = get_max<int>(first, second);
```

We can write class templates too!

- We can define class with generic types, too!
- Same syntax as a regular class, except that it is preceded by the **template** keyword and a series of template parameters enclosed in angle brackets
- It makes no difference whether the generic type is specified with keyword **class** or keyword **typename** in the template argument list (they are 100% synonyms in template declarations).

An important note

- **The entire template must be in the header file**
- We cannot separate the interface (header file) from the implementation (source file)
- This is because the templates are **compiled as required**

entry.cpp

Multiple typename identifiers

```
template <typename K, typename V>
class Entry{
    K key;
    V value;
public:
    Entry(K key, V value) :
                key{key}, value{value} {}
};
```

Specific typename identifiers

```
template <typename T, int N>
class MySet{
    T set [N];
public:
    void set_member(int index, T member);
    T get_member(int index);
};

template <typename T, int N>
void MySet<T, N>::set_member(int index, T member)
{
    set[index] = member;
}
```

set.cpp

Default typename identifiers

```
template <typename T = string, int N = 25>
class MySet{
    T set [N];
public:
    void set_member(int index, T member);
    T get_member(int index);
};

template <typename T, int N>
void MySet<T, N>::set_member(int index, T member)
{
    set[index] = member;
}
```

set2.cpp

Let’s develop a generic printing template

```
// print.h
#include <iostream>

template <typename C>
void print(const C& c)
{
    for (typename C::const_iterator it = c.begin();
         it != c.end(); ++it)
    {
        std::cout << *it << std::endl;
    }
}
```

Think about printing

- Use abstraction
- Printing is like copying
- Copying is more abstract than printing
- We can develop an algorithm to copy things from one location to another
- We can even think of printing as copying a range defined by some iterators to some ostream

A copy algorithm (step by step!)

```
void copy(int a[], size_t n, int b[])
{
    size_t i;
    for (i = 0; i < n; ++i) {
        b[i] = a[i];
    }
}
```

A copy algorithm (step 2)

```
void copy(int a[], size_t n, int b[])
{
    size_t i;
    //start at the beginning
    //while we’re not at end
    for (i = 0; i < n; ++i) {
        //do copy, increment next index
        b[i] = a[i];
    }
}
```

Our copy algorithm (step 3)

```
void copy(int* first, int* last, int* result)
{
    while (first != last) {
        *result = *first;
        result++;
        first++;
    }
}
// int a [] = { 1, 2, 3, 4, 5 };
// int b [5];
// copy(a, a + 5, b);
```

Our generic copy template (final step)

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
     OutputIterator result)
{
    while (first != last) {
        *result = *first;
         result++;
         first++;
    }
    return result;
}
```

Our generic copy template (final step+)

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
     OutputIterator result)
{
    while (first != last) {
        *result++ = *first++;
    }
     return result;
}
```

Checking if a list is in ascending order

- What if we want to ensure a list is in ascending order
- **What a great opportunity for a template!**

- Check out **ascending.cpp**

Remember functors

```
struct is_divisible_by{
    int divisor;
    is_divisible_by(int d): divisor{d} {}
    // const no change to divisor
    bool operator()(int number) const {
        return number % divisor == 0;
    }
}
is_divisible_by div5(5); // function object
div5(5); // returns true
div5(11); // returns false
```

Let's convert it to a template

```
template<class T>
struct is_div_by{
    T divisor;
    is_div_by(T d) : divisor{d} {}
    bool operator()(T n) const {
        return n % divisor == 0;
    }
};

is_div_by<int>div10(10);
std::cout << div10(10) << std::endl;
```

IN CLASS ACTIVITY

1. The Learning Hub -> Content -> Activities **"Templates In Class Activity"**