CS310

Advanced C++:
Templates and Generic Programming

Templates Overview &
Function Templates

Hüseyin Akcan
huseyin.akcan@ieu.edu.tr
Overview - The basics

- General concepts and language features of C++ templates
- Function and class templates
Why templates?

- C++ requires us to declare variables, functions, and most other kinds of entities using specific types.
- However, a lot of code looks the same for different types. This is bad programming technique!
Alternatives without templates

- If your language does not support templates you have the following alternatives:
  
  1) You can implement the same behavior again and again for each type that needs this behavior.
  2) You can write general code for a common base type such as Object or void*.
  3) You can use special preprocessors.
Each of the approaches has drawbacks

1) If you implement a behavior again and again, you reinvent the wheel. You make the same mistakes and you tend to avoid complicated but better algorithms because they lead to even more mistakes.

2) If you write general code for a common base class you lose the benefit of type checking. In addition, classes may be required to be derived from special base classes, which makes it more difficult to maintain your code.

3) If you use a special preprocessor such as the C/C++ preprocessor, you lose the advantage of formatted source code. Code is replaced by some "stupid text replacement mechanism" that has no idea of scope and types.
Case example: Swap function

- Swaps the contents of the two fields using a temp field
Case Example: Solution using Case 1

- How do you solve the problem if you need to swap values of int, double, float?

See swap1.cpp
Case 1 solution: Swap1.cpp

- Write a separate function for each type

```cpp
void my_swap(int& a, int& b){
    int tmp = a;
    a = b;
    b = tmp;
}

void my_swap(float& a, float& b){
    float tmp = a;
    a = b;
    b = tmp;
}
```
Case Example: Solution using Case 2

- How do you solve the problem if you need to swap values of int, double, float?
  1) Using void*
  2) Using common base class

See swap2_1.cpp & swap2_2.cpp
Case 2 solution(1): Swap2_1.cpp

- Solution (1): Use void* to handle unknown typenames

```cpp
void my_swap(void** pA, void** pB) {
    void* pTmp = *pA;
    *pA = *pB;
    *pB = pTmp;
}
```

This approach is unsafe, since it does not do type checking.

```cpp
float fa = 1.1;
float fb = 2.2;
cout << "fa: " << fa << " fb: " << fb << endl;
my_swap((void**) &fa, (void**) &fb);
cout << "fa: " << fa << " fb: " << fb << endl;
...

/* Undefined behavior - Runtime ERROR!!! */
my_swap((void**) &a, (void**) &fb);
cout << "a: " << a << " fb: " << fb << endl;
```

Correct usage

Incorrect usage
Case 2 solution(2): Swap2_2.cpp

- Solution (2): Use common base class and polymorphism

```cpp
class Base {
    public:
        virtual void swap(Base& other) {};
};

class A : public Base {
    public:
        A(int a) { _a = a; }
        int getA() { return _a; }
        void setA(int a) { _a = a; }

        void swap(Base& other) {
            A& _A = dynamic_cast<A&>(other);
            int _tmp = _a;
            _a = _A.getA();
            _A.setA(_tmp);
        }
    private:
        int _a;
};

void my_swap(Base& Obj1, Base& Obj2) {
    Obj1.swap(Obj2);
}```
This approach is also unsafe!
Although it does type checking during runtime it does not do type checking during compile time!
Case 3 solution: Swap3.cpp

- Use C macro definitions

```cpp
#define my_swap(type, i, j) {type t = i; i = j; j = t;}

int main(){
    int a = 1;
    int b = 2;
    cout << "a: " << a << " b: " << b << endl;
    my_swap(int,a,b);
    cout << "a: " << a << " b: " << b << endl;

    float fa = 1.1;
    float fb = 2.2;
    cout << "a: " << fa << " b: " << fb << endl;
    my_swap(float,fa,fb);
    cout << "a: " << fa << " b: " << fb << endl;
}
```

No formatted code, scope etc...
Templates

- Templates are a solution to this problem without these drawbacks.
- Because templates are language features, you have full support of type checking and scope.
- Inside the C++ standard library almost all code is template code
Function templates

- Function templates provide a functional behavior that can be called for different types.
- A function template represents a family of functions.

```cpp
template<typename T>
inline void my_swap(T& t1, T& t2){
    T temp = t1;
    t1 = t2;
    t2 = temp;
}

int main(){
    int a = 1;
    int b = 2;
    cout << "a: " << a << " b: " << b << endl;
    my_swap(a,b);
    cout << "a: " << a << " b: " << b << endl;

    float fa = 1.1;
    float fb = 2.2;
    cout << "a: " << fa << " b: " << fb << endl;
    my_swap(fa,fb);
    cout << "a: " << fa << " b: " << fb << endl;
}```
my_swap function accepts two parameters t1 and t2. The type of these parameters is left open as \texttt{template parameter T}. Template parameters must be announced with syntax of the following form:

\begin{verbatim}
template < comma-separated-list-of-parameters >

The keyword \texttt{typename} introduces a so-called \textit{type parameter}
Using function templates

```cpp
int a = 1;
int b = 2;
cout << "a: " << a << " b: " << b << endl;
my_swap(a,b);
cout << "a: " << a << " b: " << b << endl;
```

Normally, templates aren't compiled into single entities that can handle any type. Instead, different entities are generated from the template for every type for which the template is used. Thus, my_swap() is compiled for each of these different types.

The process of replacing template parameters by concrete types is called instantiation*. It results in an instance of a template.

```cpp
inline void my_swap(int& t1, int& t2){
    int temp = t1;
    t1 = t2;
    t2 = temp;
}
```

* Do not confuse this with the instantiation of an object in Object Oriented programming
Using function templates

• The mere use of a function template can trigger such an instantiation process. There is no need for the programmer to request the instantiation separately.

• Thus, templates are compiled twice:

  1) **Without instantiation**, the template code itself is checked for correct syntax. Syntax errors are discovered, such as missing semicolons.

  2) **At the time of instantiation**, the template code is checked to ensure that all calls are valid. Invalid calls are discovered, such as unsupported function calls.
Argument deduction

- When we call a function template such as max() for some arguments, the template parameters are determined by the arguments we pass.

- No automatic type conversion is allowed here. Each T must match exactly.

```cpp
template <typename T>::
inline T const& max (T const& a, T const& b);
```

max(4,7)  // OK: T is int for both arguments.
max(4,4.2) // ERROR: first T is int, second T is double.

- There are three ways to handle such an error:

  1) Cast the arguments so that they both match:

     max(static_cast<double>(4),4.2)    // OK

  2) Specify (or qualify) explicitly the type of T:

     max<double>(4,4.2)                 // OK

  3) Specify that the parameters may have different types.
Different template parameter types

- You can specify as many template parameters as you like

```cpp
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b) {
    return a < b ? b : a;
}
max(4, 4.2)  // OK, but type of first argument defines return type.
```

- Depending on the call argument order the maximum of 42 and 66.66 might be the double 66.66 or the int 66.

- Another drawback is that converting the type of the second parameter into the return type creates a new, local temporary object. As a consequence, you cannot return the result by reference. In our example, therefore, the return type has to be T1 instead of T1 const&.
Specifying return type

- You can alternatively define another parameter for the return type.
- However, template argument deduction does not match up return types, and RT does not appear in the types of the function call parameters. Therefore, RT cannot be deduced. As a consequence, you have to specify the template argument list explicitly.

```cpp
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b);
```

```cpp
max<int, double, double>(4, 4.2) // OK, but tedious.
```

- One other way to simplify is to just specialize the return type, leave the rest to the template initialization system.

```cpp
template <typename RT, typename T1, typename T2>
inline RT max (T1 const& a, T2 const& b);
```

```cpp
max<double>(4, 4.2) // OK: return type is double.
```

- In this example, the call to max<double> explicitly sets RT to double, but the parameters T1 and T2 are deduced to be int and double from the arguments.
Overloading function templates

- Like ordinary functions, function templates can be overloaded.

```cpp
// maximum of two int values.
inline int const& max (int const& a, int const& b)
{
    return a<b?b:a;
}

// maximum of two values of any type.
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a<b?b:a;
}

// maximum of three values of any type.
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return max (max(a,b), c);
}
```

See overload.cpp