CS310

“Advanced C++: Templates and Generic Programming”

Tricky Basics

Hüseyin Akcan
huseyin.akcan@ieu.edu.tr
Tricky basics

- We will see some further basic aspects of templates that are relevant to the practical use of templates
Keyword typename

- The keyword **typename** was introduced during the standardization of C++ to clarify that an identifier inside a template is a type. Consider the following example:

```cpp
template <typename T>
class MyClass {
    typename T::SubType * ptr;
    ...
};
```

- Here, the second typename is used to clarify that SubType is a type defined within class T. Thus, ptr is a pointer to the type T::SubType.

- Without typename, SubType would be considered a static member. Thus, it would be a concrete variable or object. As a result, the expression would be a multiplication of the static SubType member of class T with ptr.

  \[ T::SubType \times ptr \]
Use of keyword typename

- In general, typename has to be used whenever a name that depends on a template parameter is a type.
- A typical application of typename is the access to iterators of STL containers in template code:

```cpp
template <typename T>
void printcoll (T const& coll) {
    typename T::const_iterator pos; // iterator to iterate over coll.
    typename T::const_iterator end(coll.end()); // end position.
    for (pos=coll.begin(); pos!=end; ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```
Using this->

- For class templates with base classes, using a name x by itself is not always equivalent to this->x, even though a member x is inherited.

- In this example, for resolving the symbol exit inside foo(), exit() defined in Base is never considered. Therefore, either you have an error, or another exit() (such as the standard exit()) is called.

- As a rule of thumb, we recommend that you always qualify any symbol that is declared in a base that is somehow dependent on a template parameter with this-> or Base<T>::.

```cpp
template <typename T>:
class Base {:
    public::
        void exit(){ cout << "Base::exit() called" << endl;};
};

template <typename T>:
class Derived : Base<T> {:
    public::
        void foo() {:
            exit();   // calls external exit() or error. 
        }.
};
```

Member Templates

- Class members can also be templates. This is possible for both nested classes and member functions.
- Normally you can assign stacks to each other only when they have the same type, which implies that the elements have the same type. However, you can't assign a stack with elements of any other type, even if there is an implicit type conversion for the element types defined.
- The default assignment operator requires that both sides of the assignment operator have the same type, which is not the case if stacks have different element types.

See code in assign.cpp
Member Templates

- By defining an assignment operator as a template, however, you can enable the assignment of stacks with elements for which an appropriate type conversion is defined.

- In assignment implementation, inside the template with template parameter T, an inner template with template parameter T2 is defined.

- Note that a template assignment operator doesn't replace the default assignment operator. For assignments of stacks of the same type, the default assignment operator is still called.

See code in assign.cpp
Template template parameters

- It can be useful to allow a template parameter itself to be a class template.

- To use a different internal container for stacks, the application programmer has to specify the element type twice. Thus, to specify the type of the internal container, you have to pass the type of the container and the type of its elements again:

```cpp
Stack<int, std::vector<int>> vStack;  // integer stack that uses a vector
```

- Using template template parameters allows you to declare the Stack class template by specifying the type of the container without respecifying the type of its elements:

```cpp
stack<int, std::vector> vStack;  // integer stack that uses a vector
```
Template template parameters

- To do this you must specify the second template parameter as a template template parameter

```
template <typename T,
    template <typename ELEM> class CONT = std::deque >

class Stack { CONT<T> elems; // elements ... }
```

- The difference is that the second template parameter is declared as being a class template:

```
template <typename ELEM> class CONT
```

- The default value has changed from `std::deque<T>` to `std::deque`. This parameter has to be a class template, which is instantiated for the type that is passed as the first template parameter:

```
CONT<T> elems;
```

- CONT is used to define a class and must be declared by using the keyword class.
**Template template parameters**

- Member functions must be modified accordingly. Thus, you have to specify the second template parameter as the template template parameter.

- Template template parameters for function templates are not allowed.

```cpp
template <typename T, template <typename> class CONT>
void Stack<T, CONT>::push (T const & elem)
{
    elems.push_back(elem);    // append copy of passed elem.
}
```

See code in template.hpp & template_test.cpp
Error: default value std::deque is not compatible with the template template parameter CONT. Template template argument must be a template with parameters that exactly match the parameters of the template template parameter it substitutes. Default template arguments of template template arguments are not considered, so that a match cannot be achieved by leaving out arguments that have default values.

Cause: std::deque template of the standard library has more than one parameter: The second parameter (which describes a so-called allocator) has a default value, but this is not considered when matching std::deque to the CONT parameter.

Workaround: We can rewrite the class declaration so that the CONT parameter expects containers with two template parameters:

```cpp
template <typename T, typename ELEM, typename ALLOC = std::allocator<ELEM> >
class CONT = std::deque>
class Stack {
    private:
        CONT<T> elems; // elements
    ...
};
```

See code in template.hpp & template_test.cpp
**Zero initialization**

- For fundamental types such as int, double, or pointer types, there is no default constructor that initializes them with a useful default value. Instead, any noninitialized local variable has an undefined value.

```cpp
void foo()
{
    int x;    // x has undefined value.
    int* ptr; // ptr points to somewhere (instead of nowhere)
}
```

- If you write templates and want to have variables of a template type initialized by a default value, you have the problem that a simple definition doesn't do this for built-in types:

```cpp
template<typename T>
void foo()
{
    T x;    // x has undefined value if T is built-in type.
}
```
Zero initialization

- It is possible to call explicitly a default constructor for built-in types that initializes them with zero (or false for bool). That is, int() yields zero. As a consequence you can ensure proper default initialization even for built-in types by writing the following:

```cpp
template <typename T>
void foo()
{
  T x = T(); // x is zero (or false) if T is a built-in type.
}
```

- To make sure that a member of a class template, for which the type is parameterized, gets initialized, you have to define a default constructor that uses an initializer list to initialize the member:

```cpp
template <typename T>
class MyClass {
  private:
    T x;
  public:
    MyClass() : x() { // ensures that x is initialized even for built-in types.
    }
    ...
};
```
Write a `max` function that will work with all of the following:

```cpp
max(10, 20);  // two integers
max(10.1, 20.1) // two floating points
max(10, 20.1) // one integer, one float
max(“ten”, “twenty”) // C-style strings
String a, b;
max(a, b) // two std::Strings
max(int*, int*) // two pointers to ints
```