CS-202

C++ Templates (Pt.2)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (8:00-12:00)	
	CLASS		CLASS	
PASS Session	PASS Session			
Project DEADLINE	**Project DEADLINE**		NEW Project	

Your 8th (and Final) Project will be announced on Thursday 4/27.

7th Project Deadline is moved to Tuesday 4/25.

- 24-hrs delay after Project Deadline incurs 20% grade penalty.
- Make sure you make good use of the provided PASS Sessions !
- Past that, NO Project accepted.

Today's Topics

Template Template(s)

Member Template(s)

Template Specialization

- Specialization *vs* Overloading.

Templated Class **`friend`**(s)

Templated name Disambiguation

- Dependent-Qualified Type names – Keyword ***`typename`***.
- Explicitly-Qualified names – Keyword **`template`**.

Template(s) Compilation process

Template(s)

Remember: **Syntax**

The Templated Function:

```
template < class T >
void Swap(T & v1, T & v2);

template < class T >
void Swap(T & v1, T & v2){ T temp = v1; v1 = v2; v2 = temp; };
```

Call with implicit / explicit template parameter statement:

```
int     i1=0,         i2=1;          Swap(i1, i2);
float   f1=0.1,       f2 = 99.9;     Swap< float >(f1, f2);
Car     c1("GRAY"),   c2("WHITE");   Swap(c1, c2);
Date    d1(4,20),     d2(4,21);      Swap< Date >(1, d2);
```

Inferred / Declared Type

```
T : int
T : float
T : Car
T : Date
```

Remember: **Syntax**

The Templated Class:

```
template < class T >
class Buffer{
 public: …
  Buffer();
 private: …
  T * m_buffer;
};

template < class T >
Buffer< T > :: Buffer(){ m_buffer = new T[…]; … }
```

Instantiation with explicit template parameter statement:

```
Buffer < int > intBuffer;
Buffer < Car > carBuffer;
```

Declared Type

```
T : int
T : Car
```

The keyword `template`

Declares a family of classes / family of functions.

➢ Two alternatives:

A) Overloading the keyword `class`:

```
template < class T >
return-type tpl-func-name(parameters-list){ … }
template < class T >
class TplClassName { … };
```

B) Using the new keyword `typename`:

```
template < typename T >
return-type tpl-func-name(parameters-list){ … }
template < typename T >
class TplClassName { … };
```

The keyword `template`

The Template Template (`template< template< … > >`)

Note: There is no Template Template Template in C++.

➢ Enforced usage-case for keywords `class` & `typename`:

Given a Templated Class:

```
template <class T>
class TplClass{ public: TplClass(); …   private: T * m_t_Pt; … };
```

Creating another Templated Class, for which one Template Type is the latter:

```
template < template < typename > class ClassType , class Type >
class TplTplClass {
  public: …
  private:
    ClassType < Type > m_classType;
    …
};
```

Note: Template Template syntax needs keyword `class` to compile.

A member that is of a generalized Class Type that is itself templated for a generalized Type.

Template(s)

The keyword `template`

The Template Template (`template< template< … > >`)

➢ Utility By-Example:

2 (or more) different Templated Classes:

```
template < class T >
class ClassA {
  public: ClassA();
  private: T m_t_arr[…];
};
```

```
template < class T >
class ClassB {
  public: ClassB();
  private: T * m_t_Pt;
};
```

The Template Template Class:

```
template <template < typename > class ClassType , class Type>
class AdvancedClass {
  public: …
  private:
    ClassType < Type > m_advanced; …
};
```

Possible to create:

```
AdvancedClass< ClassA , int > intAType;
AdvancedClass< ClassB , Car > carBType;
```

`m_advanced` is of *`ClassA`* for `int` Types !
`m_advanced` is of *`ClassB`* for `Car` Types !

Template(s)

The keyword `template`

The Template Template (`template< template< … > >`)

➢ Utility By-Example:

2 (or more) different Templated Classes for element storage:

```
template < class T >
class ArrayContainer {
 public: ArrayContainer();
 private: T m_arr[…];
};
```

```
template < class T >
class NodeContainer {
 public: NodeContainer();
 private: T * m_head, * m_tail;
};
```

A Template Template Class implementing a DDS :

```
template <template < typename > class ContainerType , class Type>
class Queue { //or class Stack, or another DDS implementation
  public: …
  private:
     ClassType < Type > m_container; …
};
```

Create `Queue` variations:

```
Queue< ArrayContainer, int > intArray_q;
Queue< NodeContainer,  Car > carNode_q;
```

`m_container` is `ArrayContainer` w/ `int`s!
`m_container` is `NodeContainer` w/ `Car`s !

Template(s)

The keyword `template`

Nested / Member `template` Architecture(s).

- Templated method of a Templated Class, with separate Template Parameters.

```
template <class T>
class TplClass{
 public:
  void TplFunc();
  template <class M>
  void TplFuncTpl(M * t_arr);

  …
};
```

Nested Template Implementation (by-Example):

```
template < class T >
template < class M >
void TplClass< T > :: TplClassTpl( M * t_arr)
{
    /* T and M - mentioning implementation */
};
```

The keyword `template`

Nested / Member `template` Architecture(s).

➢ Utility By-Example:

A Templated Class that provides an assignment operator (`=`) which will be used to handle conversion assignment from (almost) any other Type **M**.

```
template < class T >
class Buffer {
 public:
   Buffer();
   template < class M >
   Buffer<T> & operator=(const Buffer<M> & other);
 private:
   T m_array[MAX_SIZE];
};
```

```
template < class T >
template < class M >
Buffer<T> & Buffer<T>::operator=( const Buffer<M> & other){
  for ( … ) {    m_array[i] = other.m_array[i]; }
  return *this;
};
```

(any) Type **T**

(any) Type **M**

Note:
Assignment operator has to be defined for **T** = **M** !

The keyword `template`

Specialization of `template`(s).

- Making distinct family members for specific Types.

Templated Class Specialization.

Syntax:

```
template < >
class TplClass < special-type >
{ … }
```

- Other members, methods, type-specific method implementations, etc.
- Altogether a different Class.

Buffer Class Template

```
template < class T >
class Buffer {
 public:
  Buffer(); …
 private:
  T m_array[ARRAY_MAX];
  int m_size;
};
```

char–specialized Buffer Class Template

```
template < >
class Buffer < char > {
 public:
  Buffer(); …
 private:
  T m_array[STRLEN_MAX];
  int m_strLength;
};
```

The keyword `template`

Specialization of `template`(s).

- Making distinct family members for specific Types.

Templated Function Specialization.

Syntax A) – Implicit Type Deduction:

```
template < >
return-type TplFunc(params){ … }
```

Syntax B) – Explicit Type Deduction:

```
template < >
return-type TplFunc<Types>(params){ … }
```

- Handles specific case of `char` value swapping.
- Implicit Type Deduction of `T:=char` by the compiler.

Function Swap Template

```
template < class T >
void Swap(T & v1, T & v2)
{
 T temp=v1; v1=v2; v2-temp;
};
```

`char`–Specialized Function Swap Template

```
template < >
void Swap(char & v1, char & v2)
{
 if ((v1…a-Z) && (v2…a-Z)){
   char temp=v1;
   v1=v2; v2-temp;
 }
};
```

Template(s)

The keyword `template`

Specialization of `template`(s).

- Making distinct family members for specific Types.

Partial Specialization → Refers to a modifier of T.

Note: Formally Function Templates are not Partially Specializing, they Overload.

Function Max Template

```
template < class T >
T Max(const T & v1,
      const T & v2){
  return (v1 < v2)? v2:v1;
};
```

Syntax:

```
template < class T >
T-mod-return TplFunc(T-mod-params)
{ … }
```

Pointer-Specialized Function Max Template

```
template < class T >
T * Max(const T * & v1,
        const T * & v2){
  return (*v1 < *v2)? v2:v1;
};
```

- Dereferences `T *`, otherwise the expression **v1<v2** evaluates relationship of memory addresses.
- Pointer-Specialized version applies `T`-reliant `operator<` on objects (has to be defined for `T`).

The keyword `template`

Specialization of `template`(s).

➢ Utility by-Example:

```
const char * s1 = "Hello";
const char * s2 = "You Fool";
cout << Max(s1, s2);
```

Function Max Template

```
template < class T >
T Max(const T & v1,
      const T & v2){
  return (v1 < v2)? v2:v1;
};
```

Compiler-made Direct Template use with `char *`

```
char * Max(const char * & v1,
           const char * & v2){
  return (v1 < v2)? v2:v1;
};
```

➢ Sorts them by the highest memory address.

Template(s)

The keyword `template`

Specialization of `template`(s).

- Utility by-Example:

```
const char * s1 = "Hello";
const char * s2 = "You Fool";
cout << Max(s1, s2);
```

Function Max Template

```
template < class T >
T Max(const T & v1,const T & v2){
  return (v1 < v2) ? v2:v1;
};
```

Pointer-Specialized Function Max Template

```
template < class T >
const T * Max(const T * & v1,
              const T * & v2){
  return (*v1 < *v2) ? v2:v1;
};
```

Note:
Partial Specialization for Pointer-modifier of `T` is necessary, otherwise the Explicit Specialization will not find an appropriate function definition.

`char *`-Specialized Function Max Template

```
template < >
const char* Max(const char* & v1,
                const char* & v2)
{
  return strcmp(v1,v2) ? v2:v1;
};
```

- `operator<` is not defined to do what we want for C-strings, hence we specialize function for `char *`-handling with `strcmp`.

The keyword `template`

Specialization of `template`(s).

- Utility by-Example:

```
const char * s1 = "Hello";
const char * s2 = "You Fool";
cout << Max(s1, s2);
```

Function Max Template

```
template < class T >
T Max(const T & v1,const T & v2){
  return (v1 < v2) ? v2:v1;
};
```

- Function Overloading is simpler.
- Compiler will always give Overloaded functions precedence over any Template.

Simpler:

`char*`–Overloaded Function Max

```
const char * Max(const char* & v1,
                 const char* & v2)
{
  return strcmp(v1,v2) ? v2:v1;
};
```

- If Overloading is possible, use it.
- If you have to, then use Specialization. Why would I ?
  a) Somewhere/someone explicitly calls a Templated Function version, e.g. `Max<char *>(s1, s2);`
  so the compiler is forced to use the Templated version.
  b) You got mixed up in Metaprogramming…

The keyword `template`

Specialization of `template`(s).

➢ Compiler rules - Specialization semantics:

A non Explicitly-Templated call of a function.

```
int * int_Pt;
func( int_Pt );
```

a) If an Overloaded function definition matches the call, it will have precedence:

```
void func(int *);
```

b) If no such match is found, Base-Templates are queried, and the "most specialized" will be used:

```
template<class T>      (a) A Base-Template
void func( T );

template<class T>      (b) A second Base-Template,
void func( T * );          (Partially-Specializes/) Overloads (a)

template <>            (c) Explicit Specialization of (b)
void func<>(int *);
```

```
template<class T>      (a) A Base-Template
void func( T );

template <>            (c) Explicit Specialization of (a)
void func<>(int *);

template<class T>      (b) A second Base-Template,
void func( T * );          (Partially-Specializes/) Overloads (a)
```

Compiler will perform: 1st Overload Resolution
2nd Specialization Lookup

The more specialized (the one that gets called).

The keyword `template`

Parameter List of `template`(s).

- Supports multiple Parameter Types & Non-Type Parameters:

```
template < class T, class U, class V, int N, char C >
return-type multi-tpl-func-name(parameters-list){ … }
template < class T, class U, class V, int N, char C >
class MultiTplClassName { … };
```

- Supports Default Parameters:

```
template < class T = int, int N = MAX_ELEMENTS >
class MultiTplClassName { … };
```

Note:
Only for Class Templates.

The keyword `template`

Function Parameter List(s) & `template`(s).

- A function parameter can be a Templated Class object:

```
template < class T >
void Sort( ArrayContainer< T > arrayContainer ){
  for (…){
    if ( arrayContainer[…] < arrayContainer[…] )
       arrayContainer[…] = …;
  }
}
```

- Type `T` might never appear in the function (not a Templated Function itself), but the function parameter is a Templated Class (`ArrayContainer<T>`) object.
- Still have to declare that the function is based off a Template.

Template(s)

The keyword template

Class friend(s) & template(s).

- A Templated Class can have friend(s) → they need to be Templated too.

```
template < class T >
class ArrayContainer {
 public:
   ArrayContainer(); …
 friend ostream & operator<<(ostream& os, const ArrayContainer<T> & a);
 private:
   T * m_arr; …
};
```

- Non-member Function is also non-Templated
- Compiler will not attempt to find code Template & create code for one to match the Class Template.

```
warning: friend declaration 'std::ostream& operator<<(std::ostream&, const
ArrayContainer<T>&)' declares a non-template function [-Wnon-template-friend]
note: (if this is not what you intended, make sure the function template has
already been declared and add <> after the function name here)
```

Compilation proceeds with warnings.

```
undefined reference to `operator<<(std::ostream&, ArrayContainer<int> const&)'
```

Linking fails.

The keyword template

Class friend(s) & template(s).

- A Templated Class can have friend(s) → they need to be Templated too.

```
template < class T >
class ArrayContainer {
 public:
   ArrayContainer(); …
 friend ostream & operator<< <> (ostream& os, const ArrayContainer<T> & a);
 private:
   T * m_arr; …
};
```

Declares the (any) Templated operator<< as a friend.

- Implementation of the Templated friend function.

```
template < class T >
ostream& operator<< (ostream & os, const ArrayContainer<T> & a)
{   for (…){ os << a[i]; } …  return os; }
```

Template(s)

The keyword `template`

Class `friend`(s) & `template`(s).

➢ Actual requirements for compiler to work:

Forward Declarations

```
template<class T> class ArrayContainer;
template<class T> ostream & operator<< (ostream & os, const ArrayContainer<T> & a);
```

Class Template

```
template < class T >
class ArrayContainer {
 public:
   ArrayContainer(); …
 friend ostream & operator<< <> (ostream & os, const ArrayContainer<T> & a);
 private:
   T * m_arr; …
};
```

Function Implementation

```
template < class T >
ostream & operator<< (ostream & os, const ArrayContainer<T> & a){
  for (…){ os << a[i]; } …
  return os;
}
```

The keyword `template`

Class `friend`(s) & `template`(s).

➢ Or the "easy" way:

Inline Declaration inside Templated Class Body.

```
template < class T >
class ArrayContainer {
 public:
   ArrayContainer(); …
 friend ostream & operator<< (ostream & os, const ArrayContainer<T> & a){
   for (…){ os << a[i]; }
   return os;
 }
 private:
   T * m_arr; …
};
```

The keyword `template`

Disambiguation of `template`(s).

➢ Dependent-Qualified Type names:

```
template < class T >
void Sort( ArrayContainer< T > arrayContainer ){
  ArrayContainer< T > :: Accessor ac;

  ac.MoveToBeginning(); T data = ac.AccessData();
  ac.MoveForward(); T data = ac.AccessData();
  …
}
```

➢ `Accessor` is a separate Class, but its actual code is not just Templated by type T.

➢ It is a Qualified & Dependent name.

Compiler needs to know T refers to a Type name.

```
typename ArrayContainer< T > :: Accessor ac;
```

The keyword `template`

Disambiguation of `template`(s).

➢ Explicitly-Qualified names:

```
class TplMethodClass {
  public: …
    template<class T>
    T tpl_method();
};

template <class U>
void func(U arg)
{
    int obj =
    arg. template member_func<int>();
}
```

Compiling Templates

Function & Class `template`(s) compilation process.

- Keep declarations (normally placed in the .h header file in any case) as well as implementation (normally placed in the .cpp source file inside A SINGLE Header (.h) file.

- Include this (.h) Header file in all places you would normally #include your Function / Class Declarations.

CS-202

Time for Questions !