CS-202

C++ Templates (Pt.1)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 9th Project will be announced today Thursday 4/18.

8th Project Deadline was this Wednesday 4/17.

- ➢ NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- ➢ Send what you have in time!

Today's Topics

Generic Programming

C++ Template(s)
- Templated Functions.
- Templated Classes.

The `template` keyword
- Usage with the overloaded `class` keyword (for Templating).
- Usage with the `typename` keyword.
- Multiple Template Parameter Types.
- Non-Type Template Parameters.

Remember: Abstraction

Abstraction in C++ (so far)

"Information Hiding", separation of code use from code implementation.

- *Data* Abstraction.
- *Control* Abstraction.

Using Classes:

```
class MyClass{
  void/int/double/class/… myClassMethod( … );
  int/double/struct/class/… m_data;
};
```

"Conceptual Types", nonconcrete, non-instantiatable parents.

- Data *Type* Abstraction.

Using Pure `virtual` Methods:

```
class ConceptClass{
  void implementConcept( … ) = 0;
};
class RealizationClass : public ConceptClass {
  void implementConcept( … ){ /* actual code */ }
};
```

Abstraction (continued)

Generic Programming

Generalizing the programming constructs of Abstraction:

- Abstracting the Abstraction.

Templates are the foundation of Generic Programming in C++.

- Enable the creation of generic interfaces through code that is non-specifically written for a particular type.

A Template is the blueprint to create:

- a Generic Function,
- a Generic Class,

(like a Class is the blueprint to create an ADT).

Utility of Generic Functions / Classes

By-Example:

- A function to swap two integers:

```
void Swap(int & v1, int & v2){
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

- Integer arguments in the Parameter List.
- Integer Local Variable.

Necessary for a working implementation.

- Function is strongly type-reliant!

Utility of Generic Functions / Classes

By-Example:

- A function to swap two float values:

```
void Swap(float & v1, float & v2){
    float temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

- Float arguments in the Parameter List.
- Float Local Variable.

Necessary for a working implementation.

- Function is strongly type-reliant!

Utility of Generic Functions / Classes

By-Example:

- A function to swap two double values:

```
void Swap(double & v1, double & v2) { /* double-explicit code */ }
```

- A function to swap two characters:

```
void Swap(char & v1, char & v2) { /* char-explicit code */ }
```

- A function to swap two Car class objects:

```
void Swap(Car & v1, Car & v2) { /* Car-explicit code */ }
```

Utility of Generic Functions / Classes

Difference between all functions is the Data Type:

- Still, a separate function definition & implementation needs to exist for every distinct variable type to be used in combination with this function.
- Otherwise (no implicit casting by the compiler will be attempted here – reasonably so):

```
void Swap(int & v1, int & v2) {
    int temp = v1;
    v1 = v2; v2 = temp;
}
int main(){
    double d1, d2;
    Swap(d1, d2);
}
```

```
error: invalid initialization of reference of
type 'int&' from expression of type 'double'
```

Remember:
Be careful if `using namespace std`.
It has a `std::swap` !

Generic Programming

Utility of Generic Functions / Classes

Difference between all functions is the Data Type:

- Still, a separate function definition & implementation needs to exist for every distinct variable type to be used in combination with this function.
- There exists a Code Re-use interface that allows handling all these together.

C++ Templates

Enable Functions and Classes that use "generic" input and types.

- Functions like `Swap(…)` only need to be written once.
- Then, they can be used with almost anything.

(Another) Use-case - Implications

Implementation of max-returning function.

```
int   Max(const int & a, const int & b );
float Max(const float & a, const float & b );
Money Max(const Money & a, const Money & b);
```

Remember:
Be careful if `using namespace std`.
It has a `std::max` !

Code looks similar for all types (where properly defined) :

```
??? Max ( ??? a, ??? b){
    if (a < b)
        return b;
    else
        return a;
}
```

Note: Has to be defined for all used types

- `int` (default is provided – OK)
- `float` (default is provided – OK)
- `class Money` (`operator<` has to be provided)

Otherwise:

```
error: no match for 'operator<' (operand
       types are 'Money' and 'Money')
```

Template(s)

Syntax

A Templated function:

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

template < class T >
return-type tpl-func-name(parameters-list){
    /* templated function implementation */
};
```

Template-based declaration has to appear before:

- ➢ Function Declaration.
- ➢ Function Implementation.

T is the "generic" type.

- ➢ Statement declares a function which works with "a Type" of variable(s) **T**.
- ➢ Hopefully for all possible Types.

Template(s)

Syntax

The Templated `max(…)` function:

```
template < class T >
T Max(const T & v1, const T &v 2);

template < class T >
T Max(const T & v1, const T & v2);
    if (v1 < v2) return v2;
    else return v1;
};
```

T is the "generic" type.

- ➢ Statement declares a function which works with "a Type" of variable(s) **T**.
- ➢ Hopefully for all possible Types.

Defining **T** as a template parameter tries to serve all possible type calls in a single function body:

- ➢ `int Max(const int &,const int &),Money Max(const Money &,const Money &)…`

Template(s)

Syntax

The Templated `Swap(…)` 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;
};
```

T is the "generic" type.

- Statement declares a function which works with "a Type" of variable(s) **T**.
- Hopefully for all possible Types.

Defining **T** as a template parameter tries to serve all possible type calls in a single function body:

- `Swap(int &,int &)`, `Swap(float &,float &)`, `Swap(Car &,Car &)` …

Template(s)

Syntax

A Templated Class:

```
template < class T >
class TplClass{
 public:
  return-type TplClassMethod(parameters-list);
};

template < class T >
return-type TplClass< T > :: TplClassMethod(parameters-list){
  /* templated class method function implementation */
};
```

Template-based declaration has to appear before:

- Class Declaration.
- Class Method(s) Implementation.

T is the "generic" type.

- Statement declares a class which works with "a Type" of variable(s) **T**.
- Hopefully for all possible Types.

Template(s)

Syntax

Templated Class Example:

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

template < class T >
T Buffer< T > :: getMax(){
  for (…){ if (…) return m_buffer[i] };
};
```

Defining **T** as a template parameter tries to create a Buffer ADT for all possible Data Types, in a single Class.

T is the "generic" type.
- Statement declares a class which works with "a Type" of variable(s) **T**.
- Hopefully for all possible Types.

Call-Syntax

Templated function call:

```
template < class T >
T Max(const T & v1, const T & v2);
```

A) Call without explicit template parameter statement:

```
int i1=0, i2=1;                 int i3 = Max(i1, i2);
double  d1=0.1, d2 = 99.9       double d3 = Max(d1, d2);
```

Deduced Type

```
T : int
T : double
```

Compiler will trt to deduce template parameter **T** from argument type(s).

Note:

```
Max(0.1, 1);
```

```
error: no matching function for call to 'Max(double, int)`
note: template argument deduction/substitution failed
note: deduced conflicting types for parameter 'T`
('double' and 'int')
```

Call-Syntax

Templated function call:

```
template < class T >
T Max(const T & v1, const T & v2);
```

B1) Call with explicit template parameter statement:

```
int i1=0, i2=1;                  int i3 = Max< int >(i1, i2);
double  d1=0.1, d2 = 99.9        double d3 = Max< double >(d1, d2);
```

Declared Type
T : int
T : double

But also:

```
int i1=0, i2=1;                  int i3 = Max< double >(i1, i2);
double d=0.1; int i=1;           int i3 = Max< int >(d, i);
```

Declared Type
T : double
T : int

Note:

- ➢ Works because parameter type `T` is explicitly declared.
- ➢ Compiles because implicit conversion is allowed with `const` (only) References to built-in types.

Call-Syntax

Templated function call:

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

B2) Call with implicit / explicit template parameter statement:

```
int i1=0, i2=1;                Swap(i1, i2)
double  d1=0.1, d2 = 99.9      Swap< double >(d1, d2);
```

Will not compile (even though type T is explicitly declared):

```
int i1=0, i2=1;              Swap< double >(i1, i2);
double d=0.1; int i=1;       Swap< int >(d, i);
```

Because:

- `T & v1`, `T & v2` are non-`const` References, since their values need to be altered by `Swap(…)`.

Call-Syntax

Templated function call (more examples):

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

Swap(bigInt, littleInt);	<int , int> → OK
Swap(myChar, myString);	<char , char *> → Not working
Swap(myString, myConstString);	<char * , const char *> → Not working
Swap("Hello", "World");	String Literals → Not working
Swap(doubleVar, floatVar);	<double , float> → Not working
Swap(Shape1, Shape2);	<*Shape* , *Shape*> → OK

Note: Templated Functions can handle any parameter Type for which the Behavior is Defined.

- ➢ Otherwise, compilation error (good scenario).
- ➢ Or (worst-case) program does compile, but doesn't work as expected!

Call Syntax

The Templated Class:

```
template < class T >
class TplClass{
 public:
  return-type TplClassMethod(parameters-list);
};

template < class T >
return-type TplClass< T > :: TplClassMethod(parameters-list){
  /* templated class method function implementation */
};
```

Template-based declaration has to appear before:

- Class Declaration.
- Class Method(s) Implementation.

Template(s)

Syntax

The Templated Class:

```
template < class T >
class Buffer{
 public: …
  Buffer();
};
template < class T >
Buffer< T > :: Buffer(){ /* default ctor code */ }
```

Instantiation with explicit template parameter statement:

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

Declared Type

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

- Expressions call default constructor of a Class **Buffer**, for which all **T**'s are interpreted as the desired type.

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

Multiple template Parameter Types.

➢ The Template Parameter List:

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

T , U , V , are the "generic" type(s).

- ➢ Statement declares a class / function which works with "the Types" of variable(s) T,U,V.
- ➢ These supposedly appear in their own right, inside the class / function body.

The keyword `template`

Multiple `template` Parameter Types.

➢ The Template Parameter List:

```
template < class T , class U >
T Min(const T & a, const U & b){
    return ( (a < b) ? a : b);
}
```

Remember:
Be careful if `using namespace std`.
It has a `std::min` !

```
int i=0;
long l = 1000;
```

```
Min< long, int >(l, i);
Min(l, i);
```

Declared / Inferred Type
`T : long`
`U : int`

```
Min< int, int >(l, i);
```

Declared Type
`T : int`
`U : int`

Each call is a different Templated function version → Different results:

```
Min(0.01,0.1);
Min<int, float>(0.01,0.1);
Min<float, int>(0.01,0.1);
```

The keyword template

Multiple template Parameter Types.

➢ Heterogeneous storage Class / Struct:

```
template < class T1 , class T2 >
class Pair {
    public:
        Pair(const T1 & x, const T2 & y);
        Pair(const Pair <T1, T2> & other);
        bool operator= (const Pair <T1, T2> & other) const;
        T1 & first();
        T2 & second();
    private:
        T1 first;
        T2 second;
};
```

Remember:
Be careful if using namespace std.
It has a std::pair (actually a struct)!

The keyword template

Multiple template Parameter Types.

➢ Heterogeneous storage Class / Struct:

```
template <class T1, class T2>
bool Pair<T1,T2>::operator==(const Pair<T1,T2>
                              & other) const {
   return m_first == other.m_first &&
          m_second == other.m_second;
}
template <class T1, class T2>
T1& Pair<T1,T2>::first() const {
   return m_first;
}
template <class T1, class T2>
T2& Pair<T1,T2>::first() const {
   return m_second;
}
```

```
template < class T1 , class T2 >
class Pair {
 public:
  Pair(const T1 & x, const T2 & y);
  Pair(const Pair<T1,T2> & other);
  bool operator=(const Pair<T1,T2>
                  &other) const;
  T1 & first();
  T2 & second();
 private:
  T1 m_first;
  T2 m_second;
};
```

The keyword `template`

Multiple `template` Parameter Types.

➢ Heterogeneous storage Class / Struct:

```
Employee johnDoe("John Doe");
Employee janeDoe("Jane Doe");
Pair<double, Employee> paycheck1(4975.5, johnDoe);
Pair<double, Employee> paycheck2(4975.5, janeDoe);
(paycheck1 == paycheck2) ? … : … ;
```

```
template < class T1 , class T2 >
class Pair {
 public:
  Pair(const T1 & x, const T2 & y);
  Pair(const Pair<T1,T2> & other);
  bool operator=(const Pair<T1,T2>
                    &other) const;
  T1 & first();
  T2 & second();
 private:
  T1 m_first;
  T2 m_second;
};
```

- ➢ Generalized Class/Struct allows making Data Pairs without writing new code to accommodate every new variation.
- ➢ Data operations can also be performed based on operator overloading.

The keyword `template`

Multiple `template` Parameter Types.

➢ Heterogeneous storage Class / Struct:

```
Pair<double, Employee> payroll[100];
payroll[0] = Pair<double, Employee>(1950.0, empl0);
…
Pair<double, Employee> topPayed(Pair <double, Employee> * pr);
  double maxSal = -1.0; int maxI = 0;
  for (int i=0; i<100; ++i)
    if (payroll[i].first() >= maxSalary ){
      maxI = i; maxSal = payroll[i].first();
    }
  return pr[i];
}
```

➢ Data Pairs enable treating heterogeneous data as a single unit.

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`

Non-Type `template` Parameter(s).

- Template Parameters List accepts regular parameters.

```
template <class T, int N>
class TplBuffer {
   public:
      T & operator[](int index);
   private:
      T  m_buffer[ N ];
};

TplBuffer  < double, 100 > dBuffer;
dBuffer[99] = 0.1;
```

Templated implementation of array size:

- Templated code is generated at compile-time by the compiler, **N** has to be constant.

Otherwise:

```
int n;
TplBuffer <double, n> dBuffer;
error: 'n' cannot appear in a constant-expression.
```

The keyword `template`

Non-Type `template` Parameter(s).

- Template Parameters List accepts regular parameters.

```
template <class T, int N>
class TplBuffer {
   public:
      T & operator[](int index);
   private:
      T  m_buffer[ N ];
};

TplBuffer  < double, 100 > dBuffer;
dBuffer[99] = 0.1;
```

Note: Make sure N is a positive integer !

[C++11]: [..] If the constant-expression (5.19) is present, it shall be an integral constant expression and its value shall be greater than zero. The constant expression specifies the bound of (number of elements in) the array. If the value of the constant expression is N, the array has N elements numbered 0 to N-1, and the type of the identifier of D is "derived-declarator-type-list array of N T". [..]

Template(s)

The keyword template

Non-Type template Parameter(s).

- Template Parameters List accepts regular parameters.

```
template<class T, int N> int tpl_arr_size( T (&arr)[N] ) {
    return N;
}
```

C++ Magic – A single function that accepts:

- An array of *any* Type **T**.
- An array of *any* integral size **N**.
- It takes it in simply by-Reference and evaluates & returns at *compile-time*:
- An `int` with the array size.

The keyword `template`

Non-Type `template` Parameter(s).

- Template Parameters List accepts regular parameters.

```
template<class T, int N> int tpl_arr_size( T (&arr)[N] ) {
    return N;
}
```

```
int arr[] = {1, 2, 3};
```

Run-time Evaluation (The program evaluates **N** by making **arr**-based function calls):

```
array_function( arr, sizeof(arr)/sizeof(arr[0]) );
```

Compile-time Deduction (The compiler infers **N** from **arr** declaration):

```
array_function( arr, tpl_arr_size(arr) );
```

Template(s)

```
template < class T >
T Max(const T & v1, const T & v2);
        if (v1 < v2) return v2;
        else return v1;
};
```

The keyword `template`

Overloading `template`(d) Functions.

➢ Even though the behavior is defined, the function performs incorrectly.

```
char * s1 = "Hello";
char * s2 = "Goodbye";
cout << Max( s1, s2 );
```

The compiler generates:

```
char * Max (const char * & a, const char * & b){
    if ( a < b )
        return b;
    else
        return a;
}
```

Will try to sort them by their Memory Address.

The keyword `template`

Overloading `template`(d) Functions.

- Create an overloaded version specifically handling `char *` variables,
  The compiler will use this instead of the Templated version.

```
char * Max(char * a, char * b)
{
    if (strcmp(a, b) < 0)
        return b;
    else
        return a;
}
```

Overloaded Function has precedence over Templated version.

Making a <Templated> Stack

Non-template(d) Version.

```
class Node
{
   public:
      Node( const int & data );
      const int & getDdata();
      void setData( const int & data );
      Node * getNext();
      void setNext( Node * next );

   private:
      int m_data;
      Node * m_next;
};
```

```
class Stack
{
   public:
      Stack();
      void push(const int & item);
      void pop();

   private:
      Node * m_top;
};
```

Making a <Templated> Stack

```
template <class T>
class Node
{
   public:
      Node( const T & data );
      const T & getData();
      void setData( const T & data );
      Node<T> * getNext();
      void setNext( Node<T> * next );

   private:
      T m_data;
      Node<T> * m_next;
};

template <class T>
Node<T> ::Node( const T & data )
{
   m_data = data;
   m_next = NULL;
}
```

```
template <class T>
const T & Node<T> ::getData()
{
   return m_data;
}

template <class T>
void Node<T> ::setData( const T & data )
{
   m_data = data;
}

template <class T>
Node<T> * Node<T> ::getNext()
{
   return m_next;
}

template <class T>
void Node<T> ::setNext( Node<T> * next )
{
   m_next = next;
}
```

Making a <Templated> Stack

```
template <class T>
class Stack
{
    public:
        Stack();
        void push(const T & value);
        void pop();

    private:
        Node<T> * m_top;

};
```

```
template <class T>
Stack<T>::Stack()
{
    m_top = NULL;
}
```

```
template <class T>
void Stack<T>::push()
{
    Node<T> * newNode = new Node<T>(value);
    newNode->setNext(m_top);
    m_top = newNode;
}
```

```
template <class T>
void Stack<T>::pop()
{
    T data = m_top->getData();
    Node<T> * temp = m_head;
    m_top = temp->getNext();
    delete temp;
}
```

Working with a Stack class <Template>

```
int main ()
{
    Stack<int> stack;

    stack.push(7);
    stack.push(8);
    stack.push(9);
    stack.push(10);

    stack.pop();
    stack.pop();
    stack.pop();
    stack.pop();


    return 0;
}
```

Note: Put declaration and implementation too in the SAME header file!

- Templates are a special case.
- More on that in next Lecture.

CS-202

Time for Questions !