ECE 2400 Computer Systems Programming
Fall 2017

Topic 15: C++ Templates

School of Electrical and Computer Engineering
Cornell University

revision: 2017-11-08-09-37

1. Function Templates

- Recall two overloaded implementations of avg

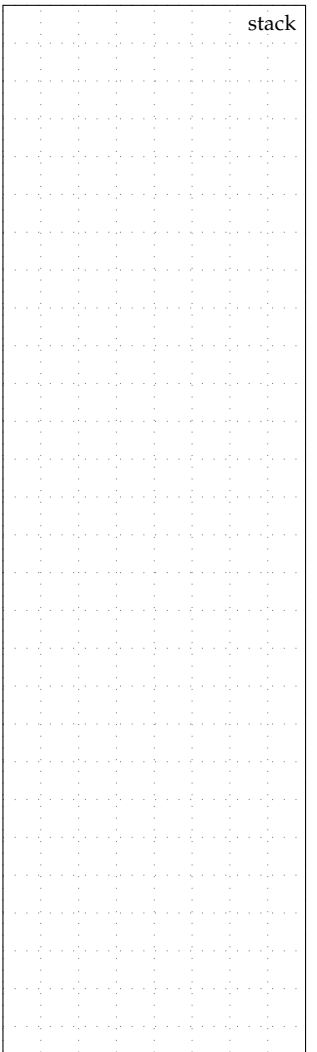
```
int avg( int x, int y )
{
  int sum = x + y;
  return sum / 2;
}

double avg( double x, double y )
{
  double sum = x + y;
  return sum / 2;
}
```

- These implementations are indentical except for the types
- Use dynamic polymorphism? but dynamic polymorphism is slow
- Instead we can use templates to implement static polymorphism

```
template < typename T >
T avg( T x, T y )
{
  T sum = x + y;
  return sum / 2;
}
```

- Does not implement a function, implements a function template
- We can instantiate the template to create a template specialization

```
1 int main( void )
2 {
3   int    a = avg<int>(5,10);
4   double b = avg<double>(5,10);
5   return 0;
6 }
```



- Compiler can infer the template argument from parameter types
- ... but be careful!

```
int main( void )
{
  int    a = avg( 5,   10   ); // will call avg<int>
  double b = avg( 5,   10   ); // will call avg<int>
  double c = avg( 5.0, 10.0 ); // will call avg<double>
  return 0;
}
```

- Can have a list of template arguments

```
template < typename T,
           typename U,
           typename V >
T avg( U x, V y )
{
  T sum = x + y;
  return sum / 2;
}
```

```
int main( void )
{
  double a = avg<double>(5,1);
  return 0;
}
```

- Template arguments can also be values

```
template < int v >
int incr( int x )
{
  return x + v;
}
```

```
int main( void )
{
  int a = 1;
  int b = incr<1>(a); // legal
  int c = 0;
  for ( int i = 0; i < 5; i++ )
    c += incr<i>(1);  // illegal!
  return 0;
}
```

2. Class Templates

- Useful to have a class that can generically store two values of potentially different types

```
struct DoubleDoublePair
{
  Pair( double a, double b ) : first(a), second(b) { }
  double first;
  double second;
}

struct CharIntPair
{
  Pair( char a, int b ) : first(a), second(b) { }
  char first;
  int second;
}
```

- Class template enables static polymorphism for classes

```
template < typename T, typename U >
struct Pair
{
  Pair( const T& a, const U& b ) : first(a), second(b) { }
  T first;
  U second;
}

int main( void )
{
  Pair pair<double,double>( 5.5, 7.5 );
  Pair pair<char,int>     ( 'a', 1   );
  return 0;
}
```

Templated Class for Statically Sized Arrays

- Implement a templated class named `Array`
- Array should contain a *statically sized* array of any type
- The type of objects in the array and the size are template parameters
- Make all member functions and member fields public for now
- Provide a default constructor and a copy constructor

3. List Using Static Polymorphism

- Dynamic polymorphic list can store any type derived from `IObject`
- Cannot store primitive types (e.g., `int`, `double`)
- Cannot store other types that do not derive from `IObject`
- Dynamic memory allocation is slow

3.1. Basic List Interface and Implementation

- Object-oriented list which stores `int`s

```
1  class List
2  {
3   public:
4    List();                          // constructor
5    ~List();                         // destructor
6    void push_front( int v );        // member function
7
8    struct Node { int value; Node* next_p; };
9
10   Node* m_head_p;                  // member field
11 };
```

- Object-oriented list which stores objects of type `T`

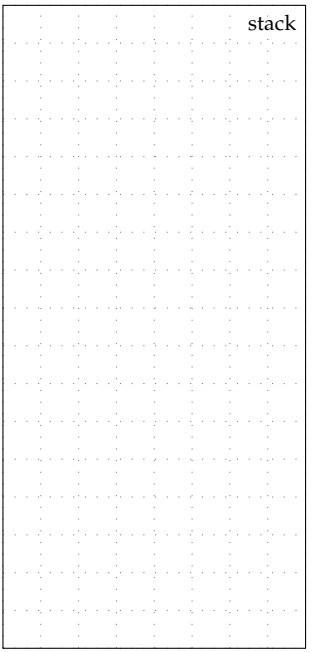
```
1  template < typename T >
2  class List
3  {
4   public:
5    List();                          // constructor
6    ~List();                         // destructor
7    void push_front( const T& v ); // member function
8
9    struct Node { T value; Node* next_p; };
10
11   Node* m_head_p;                  // member field
12 };
```

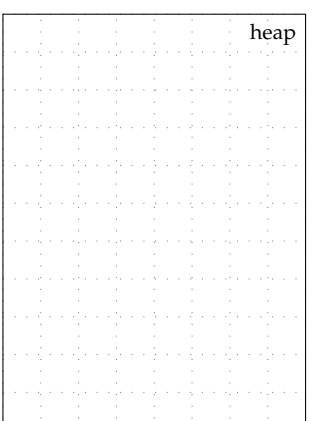
- Implementation of member functions

```
1  template < typename T >
2  List<T>::List()
3    : m_head_p(nullptr)
4  { }
5
6  template < typename T >
7  List<T>::~List()
8  {
9    Node* curr_node_p = m_head_p;
10   while( curr_node_p != nullptr ) {
11     Node* next_node_p = curr_node_p->next_p;
12     delete curr_node_p;
13     curr_node_p = next_node_p;
14   }
15   m_head_p = nullptr;
16 }
17
18 template < typename T >
19 void List<T>::push_front( const T& v )
20 {
21   Node* new_node_p   = new Node();
22   new_node_p->value  = v;           // copy by value
23   new_node_p->next_p = m_head_p;
24   m_head_p           = new_node_p;
25 }
```

- Rule of threes means we also need to declare and define a copy constructor and an overloaded assignment operator

```
1  int main( void )
2  {
3    List<int> list;
4    list.push_front( 12 );
5    list.push_front( 11 );
6    list.push_front( 10 );
7
8    Node* node_p = list.m_head_p;
9    while ( node_p != nullptr ) {
10     int value = node_p->value
11     printf( "%d\n", value );
12     node_p = node_p->next_p;
13   }
14
15   return 0;
16 }
```





3.2. Iterator-Based List Interface and Implementation

- We can use iterators to improve data encapsulation yet still enable the user to cleanly iterate through a sequence

```
1  template < typename T >
2  class List
3  {
4   public:
5
6    class Itr
7    {
8     public:
9      Itr( Node* node_p );
10     void next();
11     T&   get();
12     bool eq( Itr itr ) const;
13
14    private:
15     friend class List;
16     Node* m_node_p;
17   };
18
19   Itr begin();
20   Itr end();
21   ...
22
23  private:
24
25   struct Node
26   {
27     T     value;
28     Node* next_p;
29   };
30
31   Node* m_head_p;
32 };
```

```
1  template < typename T >
2  List<T>::Itr::Itr( Node* node_p )
3   : m_node_p(node_p)
4  { }
5
6  template < typename T >
7  void List<T>::Itr::next()
8  {
9    assert( m_node_p != nullptr );
10   m_node_p = m_node_p->next_p;
11 }
12
13 template < typename T >
14 T& List<T>::Itr::get()
15 {
16   assert( m_node_p != nullptr );
17   return m_node_p->value;
18 }
19
20 template < typename T >
21 bool List::Itr::eq( Itr itr ) const
22 {
23   return ( m_node_p == itr.m_node_p );
24 }
25
26 List::Itr List::begin() { return Itr(m_head_p); }
27 List::Itr List::end()   { return Itr(nullptr);  }
```

- Same overloaded operators as before

```
int main( void )
{
  List<int> list;
  list.push_front( 12 );
  list.push_front( 11 );
  list.push_front( 10 );

  for ( int v : list )
    std::cout << v << std::endl;
  return 0;
}
```

```
int main( void )
{
  List<double> list;
  list.push_front( 12.5 );
  list.push_front( 11.5 );
  list.push_front( 10.5 );

  for ( double v : list )
    std::cout << v << std::endl;
  return 0;
}
```

```
int main( void )
{
  typedef Pair<int,double> PairID;
  List< PairID > list;
  list.push_front( PairID(3,12.5) );
  list.push_front( PairID(4,11.5) );
  list.push_front( PairID(5,10.5) );

  for ( PairID v : list )
    std::cout << v.first << "," << v.second << std::endl;
  return 0;
}
```

3.3. Sorting with Iterators

```
1  template < typename T >
2  void List<T>::sort()
3  {
4    for ( auto itr0 = begin(); itr0 != end(); ++itr0 ) {
5      T max = *itr0;
6      for ( auto itr1 = begin(); itr1 != itr0; ++itr1 ) {
7        if ( max < *itr1 )
8          std::swap( max, *itr1 );
9      }
10     *itr0 = max;
11   }
12 }
13
14 template < typename Itr0, typename Itr1 >
15 void sort( Itr0 itr0, Itr1 itr1 )
16 {
17   for ( auto itr0 = begin(); itr0 != end(); ++itr0 ) {
18     Itr0::value_type max = *itr0;
19     for ( auto itr1 = begin(); itr1 != itr0; ++itr1 ) {
20       if ( max < *itr1 )
21         std::swap( max, *itr1 );
22     }
23     *itr0 = max;
24   }
25 }
26
27 int main( void )
28 {
29   List<int> list0;
30   list0.push_front( 12 );
31   list0.push_front( 11 );
32   list0.push_front( 99 );
33   sort( list0.begin(), list0.end() );
34   for ( int v : list0 )
35     std::cout << v << std::endl;
36 }
```

```
int main( void )
{
  List<double> list0;
  list0.push_front( 1.52 );
  list0.push_front( 11.5 );
  list0.push_front( 99.5 );

  sort( list0.begin(), list0.end() );

  for ( double v : list0 )
    std::cout << v << std::endl;

  List<IObject*> list1;
  list1.push_front( new Integer(12)   );
  list1.push_front( new Double (11.5) );
  list1.push_front( new Integer(99)   );

  // need to overload < operator for IObjects
  sort( list1.begin(), list1.end() );

  for ( IObject* obj_p : list1 )
    std::cout << obj_p->to_str() << std::endl;

  Vector<int> vec;
  vec.push_front( 12 );
  vec.push_front( 11 );
  vec.push_front( 99 );

  sort( vec.begin(), vec.end() );

  for ( int v : vec )
    std::cout << v << std::endl;

  return 0;
}
```

4. Drawing Framework w/ Dynamic & Static Polymorphism

```
class Group
{
  void add( const IShape& shape )
  {
    m_shapes.push_back( shape );
  }

  void translate( double x_offset, double y_offset )
  {
    for ( IObject* obj_p : m_shapes ) {
      IShape* shape_p = dynamic_cast<Shape*>(obj_p);
      shape->translate( x_offset, y_offset );
    }
  }

  ...

 private:
  List m_shapes;
};
```

```
class Group
{
  ~Group()
  {
    for ( IShape* shape_p : m_shapes )
      delete shape_p;
    m_shapes.clear();
  }

  void add( const IShape& shape )
  {
    IShape* shape_p = shape->clone();
    m_shapes.push_back( shape_p );
  }

  void translate( double x_offset, double y_offset )
  {
    for ( IShape* shape_p : m_shapes )
      shape_p->translate( x_offset, y_offset );
  }

  ...

 private:
  List<IShape*> m_shapes;
};
```