ECE 2400 Computer Systems Programming Fall 2021

Topic 14: Generic Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2021-08-28-18-25

zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

- Programming is organized around algorithms and data structures where *generic types* are specified upon instantiation as opposed to definition
- C can (partially) support generic programming through awkward use of the preprocessor and/or `void*` pointers
- C++ adds new syntax and semantics to elegantly support generic programming

```
#define SPECIALIZE_LIST_T( T )                                          \
                                                                        \
typedef struct                                                          \
{                                                                       \
  /* implementation specific */                                         \
}                                                                       \
list_ ## T ## _t;                                                       \
                                                                        \
void list_ ## T ## _construct  ( list_ ## T ## _t* this );              \
void list_ ## T ## _destruct   ( list_ ## T ## _t* this );              \
void list_ ## T ## _push_front ( list_ ## T ## _t* this, T v );         \
void list_ ## T ## _reverse    ( list_ ## T ## _t* this );

SPECIALIZE_LIST_T( int   )
SPECIALIZE_LIST_T( float )
```

1. C++ Function Templates

- Two implementations of `avg`

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

```
float avg_float( float x, float y )
{
  float 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;
}
```

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

```
template <>
float avg<float>( float x, float y )
{
  float 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

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

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

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

int main( void )
{
  int   a = avg<int>(5,10);
  float b = avg<float>(5,10);
  return 0;
}
```

stack

Develop a generic `contains` algorithm

Develop a generic `contains` function that takes as input an array (`arr`), the size of that array (`size`), and a value to search for in the array. The return type of the function is `bool`. It should return `true` if the value is contained in the array and `false` otherwise. The function should be generic across the type of values stored in the array. In other words, the function should work for arrays of `int`s, arrays of `float`s, etc. *Hint: Develop a version of the algorithm specialized for `int`s and then make it generic.*

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

```
int main( void )
{
  int   a = avg( 5,    10   ); // will call avg<int>
  float b = avg( 5,    10   ); // will call avg<int>
  float c = avg( 5.0f, 10.0f ); // will call avg<float>
  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 )
{
  float a = avg<float,int,int>(5,1);
  float b = avg<float>(5,1); // cannot infer from return type
  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. C++ Class Templates

- A class to generically store two values of potentially different types

```
struct PairFloatFloat
{
  PairFloatFloat( float a, float b )
    : first(a), second(b) { }
  float first;
  float second;
}

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

- Class templates enable 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<float,float> pair( 5.5, 7.5 );
  Pair<char,int>    pair( 'a', 1   );
  return 0;
}
```

3. Dynamic vs. Static Polymorphism

Dynamic Polymorphism

```


int calc_pts( const IPiece& p0,
              const IPiece& p1 )
{
  int pts0 = p0.get_pts();
  int pts1 = p1.get_pts();
  return pts0 + pts1;
}

int main( void )
{
  Pawn p('a',2);
  Rook r('h',3);
  int sum = calc_pts( p, r );
  return 0;
}
```

- Only a single version of `calc_pts` is compiled
- State diagram includes run-time type information (implicit type fields)
- Run-time type information used for dynamic dispatch
- Slower performance, more space usage

Static Polymorphism

```
template < typename T,
           typename U >
int calc_pts( const T& p0,
              const U& p1 )
{
  int pts0 = p0.get_pts();
  int pts1 = p1.get_pts();
  return pts0 + pts1;
}

int main( void )
{
  Pawn p('a',2);
  Rook r('h',3);
  int sum = calc_pts( p, r );
  return 0;
}
```

- Many versions (template specializations) of `calc_pts` are compiled
- State diagram does not include any run-time type information
- Compile-time type information used for static dispatch
- Faster performance, less space usage

Flexibility of Dynamic Polymorphism

```
int calc_pts( IPiece** pieces,
              int n )
{
  int sum = 0;
  for ( int i=0; i<n; i++ )
   sum += pieces[i]->get_pts();
  return sum;
}

int main( void )
{
  Pawn p('a',2);
  Rook r('h',3);

  IPiece* pieces[2];
  pieces[0] = &p;
  pieces[1] = &r;

  int sum
    = calc_pts( pieces, 2 );
  return 0;
}
```

- Types must inherit from an abstract base class
- Does not work well with primitive types
- Can easily mix *different* concrete types
- In general, more flexible

Flexibility of Static Polymorphism

```
template < typename T >
int calc_pts( T** pieces,
              int n )
{
  int sum = 0;
  for ( int i=0; i<n; i++ )
   sum += pieces[i]->get_pts();
  return sum;
}

int main( void )
{
  Pawn p0('a',2);
  Pawn p1('h',3);

  Pawn* pieces[2];
  pieces[0] = &p0;
  pieces[1] = &p1;

  int sum
    = calc_pts( pieces, 2 );
  return 0;
}
```

- Types must adhere to a given "concept"
- Works well with primitive types (if they adhere to concept)
- Cannot easily mix *different* concrete types
- In general, less flexible

4. Generic Lists

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

4.1. Singly Linked List Interface

- Object-oriented list which stores `int`s

```
class SListInt
{
 public:
  SListInt();                        // constructor
  ~SListInt();                       // destructor
  void push_front( int v );          // member function
  void reverse();                    // member function

  // implementation specific
};
```

- Generic list which stores objects of type `T`

```
template < typename T >
class SList
{
 public:
  SList();                           // constructor
  ~SList();                          // destructor
  void push_front( const T& v ); // member function
  void reverse();                    // member function

  // implementation specific
};
```

- C++ rule of three means we also need to declare and define a copy constructor and an overloaded assignment operator

4.2. Singly Linked List Implementation

- Corresponding implementation for an object-oriented list

```

class SListInt
{
 public:
  SListInt();
  ~SListInt();
  void push_front( int v );
  void reverse();

  struct Node
  {
    int   value;
    Node* next_p;
  };

  Node* m_head_p;
};
```

- Corresponding implementation for a generic list

```
template < typename T >
class SList
{
 public:
  SList();
  ~SList();
  void push_front( const T& v );
  void reverse();

  struct Node
  {
    T     value;
    Node* next_p;
  };

  Node* m_head_p;
};
```

- Implementation for an object-oriented list

```

SListInt::ListInt()
{
  m_head_p = nullptr;
}


void SListInt::push_front( int v )
{
  Node* new_node_p
    = new Node;
  new_node_p->value  = v;
  new_node_p->next_p = m_head_p;
  m_head_p           = new_node_p;
}


SListInt::~SListInt()
{
  while ( head_p != nullptr ) {
    Node* temp_p
      = head_p->next_p;
    delete head_p;
    head_p = temp_p;
  }
}
```

- Implementation for a generic list

```
template < typename T >
SList<T>::SList()
{
  m_head_p = nullptr;
}

template < typename T >
void SList<T>::push_front( const T& v )
{
  Node* new_node_p
    = new Node;
  new_node_p->value  = v;
  new_node_p->next_p = m_head_p;
  m_head_p           = new_node_p;
}

template < typename T >
SList<T>::~SList()
{
  while ( head_p != nullptr ) {
    Node* temp_p
      = head_p->next_p;
    delete head_p;
    head_p = temp_p;
  }
}
```

```
template <>
SList<int>::SList()
{
  m_head_p = nullptr;
}

template <>
void SList<int>::push_front(
  const int& v )
{
  Node* new_node_p
    = new Node;
  new_node_p->value  = v;
  new_node_p->next_p = m_head_p;
  m_head_p           = new_node_p;
}

int main( void )
{
  SList<int> lst;
  lst.push_front( 12 );
  lst.push_front( 11 );
  lst.push_front( 10 );

  Node* node_p = lst.m_head_p;
  while ( node_p != nullptr ) {
    int value = node_p->value
    node_p = node_p->next_p;
  }

  return 0;
}
```

stack

heap

4.3. 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

```
template < typename T >
class SList
{
 public:

  class Itr
  {
   public:
    Itr( Node* node_p );
    void next();
    T&   get();
    bool eq( Itr itr ) const;

   private:
    Node* m_node_p;
  };

  Itr begin();
  Itr end();
  ...

 private:

  struct Node
  {
    T     value;
    Node* next_p;
  };

  Node* m_head_p;
};
```

```
template < typename T >
SList<T>::Itr::Itr( Node* node_p )
 : m_node_p(node_p)
{ }

template < typename T >
void SList<T>::Itr::next()
{
  assert( m_node_p != nullptr );
  m_node_p = m_node_p->next_p;
}

template < typename T >
T& SList<T>::Itr::get()
{
  assert( m_node_p != nullptr );
  return m_node_p->value;
}

template < typename T >
bool SList::Itr::eq( Itr itr ) const
{
  return ( m_node_p == itr.m_node_p );
}

SList::Itr SList::begin() { return Itr(m_head_p); }
SList::Itr SList::end()   { return Itr(nullptr);  }
```

- Same overloaded operators as before for `++`, `*`, `!=`

```
int main( void )
{
  SList<int> lst;
  lst.push_front( 12 );
  lst.push_front( 11 );
  lst.push_front( 10 );

  for ( int v : lst )
    std::printf( "%d\n", v );
  return 0;
}
```

```
int main( void )
{
  SList<float> lst;
  lst.push_front( 12.5 );
  lst.push_front( 11.5 );
  lst.push_front( 10.5 );

  for ( float v : lst )
    std::printf( "%f\n", v );
  return 0;
}
```

```
int main( void )
{
  typedef Pair<int,float> PairIF;
  SList< PairIF > lst;
  lst.push_front( PairIF(3,12.5) );
  lst.push_front( PairIF(4,11.5) );
  lst.push_front( PairIF(5,10.5) );

  for ( PairIF v : lst )
    std::printf( "%d,%f\n", v.first, v.second );
  return 0;
}
```

5. More Complex Generic Programming

- We could add a `contains` member function to our generic list
- Would only be generic across any type stored in the list
- Need to add `contains` member function to every data structure
- We can use generic programming to create a stand-alone algorithm to work across *any* data structure (and *any* types stored in those data structures) that adheres the concept of a sequence
  - sequences have `begin`, `end`, iterators
- Generic `contains` algorithm for *any* sequence

```
template < typename S,
           typename T >
bool contains( const S& seq,

               const T& v )
{
  auto itr = seq.begin();
  while ( itr != seq.end() ) {
    if ( *itr == v )
      return true;
    ++itr;
  }
  return false;
}
```

```
template < typename Itr,
           typename T >
bool contains( Itr begin,
               Iter end,
               const T& v )
{
  auto itr = begin;
  while ( itr != end ) {
    if ( *itr == v )
      return true;
    ++itr;
  }
  return false;
}
```

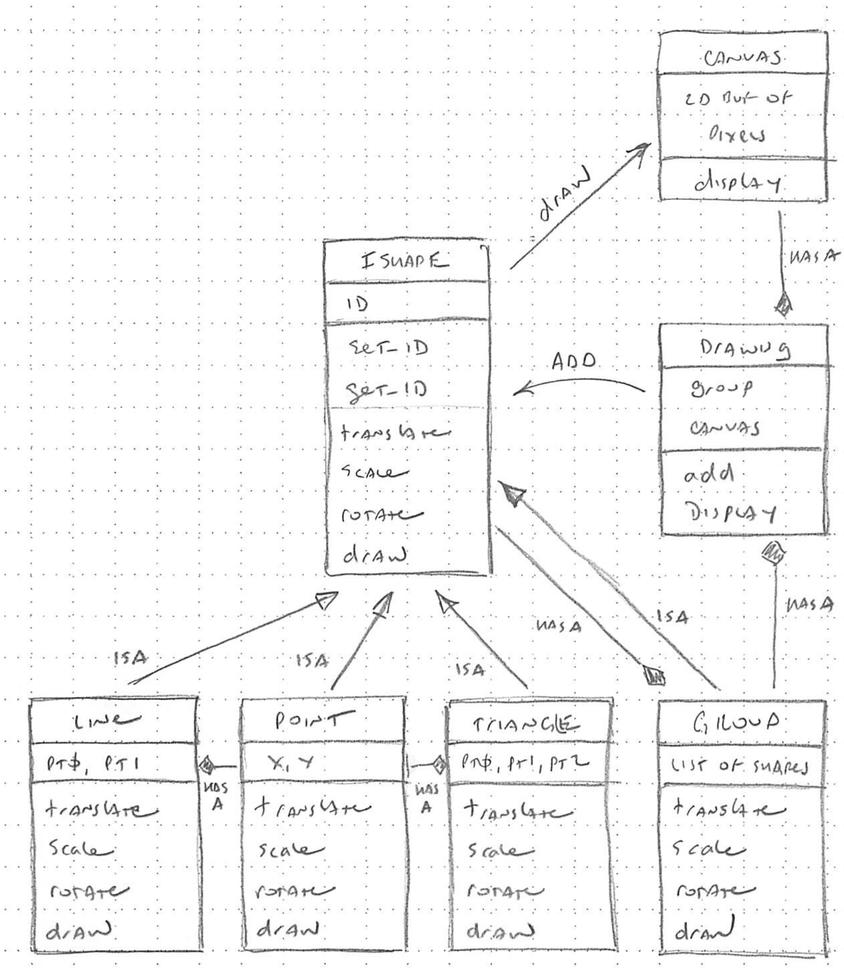
- Using generic algorithms that operate on data structures

```
SList<int> lst0;
lst0.push_front( 12 );
lst0.push_front( 11 );
lst0.push_front( 10 );
bool a =
  contains( lst0, 11 );


SList<float> lst1;
lst1.push_front( 12.5 );
lst1.push_front( 11.5 );
lst1.push_front( 10.5 );
bool b =
  contains( lst1, 11.5 );


BVector<int> vec;
vec.push_front( 12 );
vec.push_front( 11 );
vec.push_front( 10 );
bool c =
  contains( vec, 11 );


// compile time error!
bool d =
  contains( vec, Point(1,2) );
```

- Using generic algorithms that operate on iterators

```
SList<int> lst0;
lst0.push_front( 12 );
lst0.push_front( 11 );
lst0.push_front( 10 );
bool a =
  contains( lst0.begin(),
            lst0.end(), 11 );

SList<float> lst1;
lst1.push_front( 12.5 );
lst1.push_front( 11.5 );
lst1.push_front( 10.5 );
bool b =
  contains( lst1.begin(),
            lst1.end(), 11.5 );

BVector<int> vec;
vec.push_front( 12 );
vec.push_front( 11 );
vec.push_front( 10 );
bool c =
  contains( vec.begin(),
            vec.end(), 11 );

// compile time error!
bool d =
  contains( vec.begin(),
            vec.end(),
            Point(1,2) );

int a[] = { 10, 11, 12 };
bool d =
  contains( a, a+3, 11 );
```

6. Drawing Framework Case Study



- Use a statically sized array of `IShape` pointers
- `Group` must do all of the dynamic memory management

```
class Group : public IShape
{
 public:
  ...

  ~Group()
  {
    for ( int i = 0; i < m_shapes_size; i++ )
      delete m_shapes[i];
  }

  void add( const IShape& shape )
  {
    assert( m_shapes_size < 16 );
    m_shapes[m_shapes_size] = shape.clone();
    m_shapes_size++;
  }

  void translate( double x_offset, double y_offset )
  {
    for ( int i = 0; i < m_shapes_size; i++ )
      m_shapes[i]->translate( x_offset, y_offset );
  }

 private:
  int     m_shapes_size;
  IShape* m_shapes[16];
};
```

Using Dynamic Polymorphic List

- Use a dynamic polymorphic list to store `IShapes`
- Make sure `IShape` inherits from `IObject`
- `Group` does not do any dynamic memory management

```
class Group
{
 public:
  ...

  ~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<IShape*>(obj_p);
      shape->translate( x_offset, y_offset );
    }
  }

  ...

 private:
  ListIObj m_shapes;
};
```

Using Static Polymorphic List

- Use a static polymorphic list to store `IShape` pointers
- `Group` now has to handle some of the dynamic memory management

```
class Group
{
 public:
  ...

  ~Group()
  {
    for ( IShape* shape_p : m_shapes )
      delete shape_p;
  }

  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;
};
```