Parametric polymorphism

Michael R. Nowak
Texas A&M University

Motivation

- C++ requires us to declare variables and functions using specific types
- However, a lot of code looks the same for different types

Motivation

- For instance, consider the selection sort algorithm, where we repeatedly select the next smallest element in a container and swap it into the correct location in that container:

```
1. Visit each element in the container in order
   ("left-to-right")
   a. Compare the current element to each element to
      its right, while maintaining the index of the
      smallest element observed so far
   b. Once you've found the smallest element, swap
      the element at that index with the current
      element
2. Continue this process until you've visited each
   element in the container.
```

Motivation

- We could write a selection sort for a vector<int> v, and easily tailor our solution to sort a vector<char> v:

```
// selection sort algorithm for vector<int>
for ( int i = 0 ; i < v.size() ; ++i ) {
    int smallest = i;
    for ( int j = i + 1 ; j < v.size() ; ++j ) {
        if ( v.at(smallest) > v.at(j) )
            smallest = j;
    }
    int temp = v.at(i);
    v.at(i) = v.at(smallest);
    v.at(smallest) = temp;
}
```

```
// selection sort algorithm for vector<char>
for ( int i = 0 ; i < v.size() ; ++i ) {
    int smallest = i;
    for ( int j = i + 1 ; j < v.size() ; ++j ) {
        if ( v.at(smallest) > v.at(j) )
            smallest = j;
    }
    char temp = v.at(i);
    v.at(i) = v.at(smallest);
    v.at(smallest) = temp;
}
```

Motivation

- We could write a generic template for this algorithm, and then simply substitute in for type T as needed to create specializations of our algorithm to accommodate vector<T>
- If we had this template saved somewhere, we could construct specializations as needed by copying it into our code, and making the necessary substitutions for T

```
// selection sort algorithm for vector<T>
for ( int i = 0 ; i < v.size() ; ++i ) {
    int smallest = i;
    for ( int j = i + 1 ; j < v.size() ; ++j ) {
        if ( v.at(smallest) > v.at(j) )
            smallest = j;
    }
    T temp = v.at(i);
    v.at(i) = v.at(smallest);
    v.at(smallest) = temp;
}
```

Function templates

- In C++, function templates allow generic behavior to be encapsulated inside a function and then called for different types
  - The representation of such functions is almost identical to the functions that we've talked about to this point, with the exception the types of the parameters are left open as a template parameters
  - For instance, to parameterize the definition of a function that returns the minimum valued object of two objects, we would write:

```
template<typename T> T min (T a, T b)
{
    return (b > a) ? a : b;
}
```

Defining a function template

```
template<typename T> T min (T a, T b)
{
    return (b > a) ? a : b;
}
```

- We use the keyword `template`, followed by the type parameters that we'd like to announce inside angled brackets
- The keyword `typename` introduces a type parameter; here, the type parameter is identified by `T`
  - `T` represents an arbitrary type that is determined by the caller when the caller calls the function
  - Any type can be used as long as it has the operations used in the template defined; here `T` must support `operator>`

Using a function template

- When we invoke min with arguments of type `i`, an instance of the template is created, with the template parameter T being replaced by type i
  - This process of replacing template parameters by concrete types is called instantiation
  - To trigger the instantiation process, we simply invoke the function with the desired arguments:
    - For instance, invoking the min function template with double as template parameter T, has the same semantics of calling the following code:

```
double min (double a, double b)
{
    return (b > a) ? a : b;
}
```

Using a function template

(slide intentionally left blank)

Template argument deduction

- When we call a function template for some arguments, the template parameters are determined by the type of the arguments that we pass
  - If we pass two objects of the same type to our min function, the compiler will conclude that T is of that type

```
min(2,4) // T is deduced as an int
min(2.2, 4.4) // T is deduced as a double
min('a', 's') // T is deduced as a char
min("a", "s") // T is deduced as a char*
```

  - However, if we passed two objects of different type to our min function, the compiler would be unable to deduce what type T is

```
min(2, 2.4) // ERROR: T cannot be deduced as both an int and a double
min(2.4, 2) // ERROR: T cannot be deduced as both a double and an int
min(2, 'a') // ERROR: T cannot be deduced as both an int and a char
```

Template argument deduction

```
min(2, 2.4) // ERROR: T cannot be deduced as both an int and a double
min(2.4, 2) // ERROR: T cannot be deduced as both a double and an int
min(2, 'a') // ERROR: T cannot be deduced as both an int and a char
```

- We can handle these errors by either:
  1. Casting the arguments so that they are of the same type:
     ```
     min(2, static_cast<int>(2.4))
     ```
  2. Explicitly stating what type T should be, thus preventing the compiler from attempting to deduce the type of T:
     ```
     min<double>(2, 2.4)
     ```
  3. Specifying in our function template definition that the parameters may be of different types and then letting the compiler figure out the return type:
     ```
     template<typename T1, typename T2> auto min (T1 a, T2 b)
     {
         return (b > a) ? a : b;
     }
     ```

Templates and separate compilation

- For each template instantiation, the compiler generates specific code for that instantiation
  - If you have N different kinds of instantiations for class/function, you will have N different copies of code
- Recall that C++ uses separate compilation to compile multiple translation units; i.e., compiler operates on a single translation unit at a time
  - When we #include a header file, we bring the contents of that file into our source file
  - The implementation details are in the cpp file, which our source file doesn't have access to until we link things together
  - However, when using templates, we need to generate code for each instantiation at compile-time, but we don't have access to the implementation at compile time
  - What to do?

Templates and separate compilation

- Templates must be fully defined in each translation unit
  - There are many different ways to approach this problem
  - *For this class*, you will write templated class/function implementation details in the header file

Parameterizing a function : before

Max.h

```
#ifndef MAX_H
#define MAX_H

int const& max(int const& a,
               int const& b);

#endif
```

Max.cpp

```
#include "Max.h"

int const& max(int const& a, int const& b) {
    return (a < b) ? b : a;
}
```

Parameterizing a function : after

Max.h

```
#ifndef MAX_H
#define MAX_H

template<typename T>
T const& max(T const& a, T const& b)
{
    return (a < b) ? b : a;
}

#endif
```

Class templates

- Classes can also be parameterized by one or more types
  - Container classes, such as vector, are a typical example of this feature, by leaving the element type open as a template parameter
- Parameterization of types by types (and integers)

```
template<class T, int N> class Stack{ /* … */ };
template<class T, int N> void Stack<T,N>::push(T ele) { /* … */ }
```

Template specializations (instantiations)

```
// for a class template, you specify the template arguments:
Buffer<char,1024> buf; // for buf, T is char and N is 1024
```

Class templates

- To provide some exposure to class templates, let's implement a stack as a parameterized class
- From this class template, we would like to instantiate stack that are specialized to hold elements of a specific type:

```
Stack<double>  // a stack of doubles
Stack<int>    // a stack of int
Stack<char*>  // vector of pointers to char
```

Declaration of class templates

- Before the class declaration, you must declare one or more type parameters; T is conventionally used as the identifier:

```
template<typename T> class Stack {
   //...
};
```

- Inside the class template, T is used like any other type in the declaration of members and/or functions
- The use of the class name Stack inside the class template represents the instantiated class with its template parameters as arguments

```
template<typename T> class Stack {
public:
   Stack();                           // default constructor for Stack<T>
   Stack (Stack const&);              // copy constructor for Stack<T>
   Stack& operator= (Stack const&);   // assignment operator for Stack<T>
};
```

Definition of member functions

- The type of the class is Stack<T>, where T is the template parameter; therefore, this type must be used in all declarations where the template arguments cannot be deduced
- Therefore, when defining a member function outside of the class, we must provide Stack<T> as type in the fully qualified name

```
template<typename T>
void Stack<T>::push(T c)
{
    if (top == max_size) throw Overflow();
    v[top] = c;
    top += 1;
}
```

Using a class template

- By declaring type `Stack<i>`, `i` is "substituted" for `T` everywhere it appears inside the class template
  - This process of replacing template parameters by concrete types is called template specialization (aka instantiation)
  - To instantiate a class template, *you must specify the template arguments to the template parameters explicitly*

Parameterize with element type : before

Stack.h

```
#ifndef STACK_H
#define STACK_H

class Underflow{};
class Overflow{};

class Stack {
public:
    void push(char c);
    char pop();
private:
    static constexpr int max_size = 200;
    char v[max_size];
    int top = 0;
};

#endif
```

Stack.cpp

```
#include "Stack.h"

void Stack::push(char c) {
    if (top == max_size) throw Overflow();
    v[top] = c;
    top += 1;
}

char Stack::pop() {
   if (top < 0) throw Underflow();
   top -= 1;
   return v[top];
}
```

Parameterize with element type : after

Stack.h

```
#ifndef STACK_H
#define STACK_H

class Underflow{};
class Overflow{};

template<typename T>
class Stack {
public:
    void push(T c);
    T pop();
private:
    static constexpr int max_size = 200;
    T v[max_size];
    int top = 0;
};
```

```
template<typename T>
void Stack<T>::push(T c) {
    if (top == max_size) throw Overflow();
    v[top] = c;
    top += 1;
}

template<typename T>
T Stack<T>::pop() {
   if (top < 0) throw Underflow();
   top -= 1;
   return v[top];
}

#endif
```

STL `vector` parameterization

```
// an almost real vector of Ts:
template<typename T> class vector {
  // …
};
vector<double> vd;           // T is double
vector<int> vi;                     // T is int
vector<vector<int>> vvi;            // T is vector<int>
                             //           in which T is int
vector<char> vc;             // T is char
vector<double*> vpd;         // T is double*
vector<vector<double>*> vvpd;       // T is vector<double>*
                             //           in which T is double
```

Basically, `vector<double>` is

```
// an almost real vector of doubles:
class vector {
  int sz;                // the size
  double* elem;  // a pointer to the elements
  int space;             // size+free_space
public:
  vector() : sz(0), elem(0), space(0) { }         // default constructor
  explicit vector(int s) :sz(s), elem(new double[s]), space(s) { } //
  constructor
  vector(const vector&);                  // copy constructor
  vector& operator=(const vector&);               // copy assignment
  ~vector() { delete[ ] elem; }                   // destructor

  double& operator[ ] (int n) { return elem[n]; }  // access: return reference
  int size() const { return sz; }                 // the current size

  // …
};
```

Basically, **vector<char>** is

```
// an almost real vector of chars:
class vector {
  int sz;                   // the size
  char* elem;               // a pointer to the elements
  int space;                // size+free_space
public:
  vector() : sz{0}, elem{0}, space{0} { }          // default constructor
  explicit vector(int s) :sz{s}, elem{new char[s]}, space{s} { } //
  constructor
  vector(const vector&);                   // copy constructor
  vector& operator=(const vector&);                // copy assignment
  ~vector() { delete[ ] elem; }                    // destructor

  char& operator[ ] (int n) { return elem[n]; }    // access: return reference
  int size() const { return sz; }                  // the current size

  // …
};
```

Basically, **vector<T>** is

```
// an almost real vector of Ts:
template<typename T> class vector { // read "for all types T" (just like in
  math)
  int sz;                   // the size
  T* elem;                  // a pointer to the elements
  int space;                // size+free_space
public:
  vector() : sz{0}, elem{0}, space{0};             // default constructor
  explicit vector(int s) :sz{s}, elem{new T[s]}, space{s} { }       //
  constructor
  vector(const vector&);                   // copy constructor
  vector& operator=(const vector&);                // copy assignment
  vector(const vector&&);                  // move constructor
  vector& operator=(vector&&);             // move assignment

  ~vector() { delete[ ] elem; }                    // destructor


  // …
};
```

Basically, **vector<T>** is

```
// an almost real vector of Ts:
template<typename T> class vector { // read "for all types T" (just like in math)
  int sz;                   // the size
  T* elem;                  // a pointer to the elements
  int space;                // size+free_space
public:
  // … constructors and destructors …

  T& operator[ ] (int n) { return elem[n]; }       // access: return reference
  int size() const { return sz; }                  // the current size

  void resize(int newsize);                        // grow
  void push_back(double d);                        // add element

  void reserve(int newalloc);                      // get more space
  int capacity() const { return space; }           // current available space

  // …
};
```

Templates

- Problems ("there is no free lunch")
  - Poor error diagnostics
    - Often spectacularly poor (but getting better in C++11; much better in C++14)
  - Delayed error messages
    - Often at link time
  - All templates must be fully defined in each translation unit
    - (So place template definitions in header files
- Recommendation
  - Use template-based libraries
    - Such as the C++ standard library
      - *E.g.*, **vector**, **sort()**
      - Soon to be described in some detail
  - Initially, write only very simple templates yourself
    - Until you get more experience