Generic programming – Templates

(Lecture starts at 9.15 today)

6 Generic Programming – Templates

Introduction to generic programming

- So far concentrated on definitions of objects as means of abstraction
- Next: Abstracting algorithms to be independent of the type of data they work with
- Naïve – max()
  - Integer implementation

```
// Maximum of two values
int max(int a, int b) {
   return (a>b) ? a : b ;
}
```

  - (Naïve) real-life use

```
int m = 43, n = 56 ;
cout << max(m,n) << endl ; // displays 56 (CORRECT)

double x(4.3), y(5.6) ;
cout << max(x,y) << endl ; // displays 5 (INCORRECT)
```

Generic algorithms – the max() example

- First order solution – function overloading
  - Integer and float implementations

```
// Maximum of two values
int max(int a, int b) {
   return (a>b) ? a : b ;
}

// Maximum of two values
float max(float a, float b) {
   return (a>b) ? a : b ;
}
```

  - (Naïve) real-life use

```
int m = 43, n = 56 ;
cout << max(m,n) << endl ; // displays 56 (CORRECT)

double x(4.3), y(5.6) ;
cout << max(x,y) << endl ; // displays 5.6 (CORRECT)
```


Generic algorithms – the template solution

- Overloading solution works but not elegant
  - Duplicated code (always a sign of trouble)
  - We need to anticipate all use cases in advance
- C++ solution – a **template** function

```
template<class TYPE>
TYPE max(const TYPE& a, const TYPE& b) {
    return (a>b) ? a : b ;
}


int m = 43, n = 56 ;
cout << max(m,n) << endl ; // displays 56 (CORRECT)

double x(4.3), y(5.6) ;
cout << max(x,y) << endl ; // displays 5.6 (CORRECT)
```

Basics of templates

- A template function is function or algorithm for a generic TYPE
  - Whenever the compiler encounter use of a template function with a given TYPE that hasn't been used before the compiler will instantiate the function for that type

```
template<class TYPE>
TYPE max(const TYPE& a, const TYPE& b) {
    return (a>b) ? a : b ;
}

int m = 43, n = 56 ;
// compiler automatically instantiates max(int&, int&)
cout << max(m,n) << endl ; // displays 56 (CORRECT)

double x(4.3), y(5.6) ;
// compiler automatically instantiates max(float&, float&)
cout << max(x,y) << endl ; // displays 5.6 (CORRECT)
```


Basics of templates – assumptions on TYPE

- A template function encodes a generic algorithm but not a universal algorithm
  - TYPE still has to meet certain criteria to result in proper code
  - For example:

```
template<class TYPE>
TYPE max(const TYPE& a, const TYPE& b) {
    return (a>b) ? a : b ;
}
```

    assumes that `TYPE.operator>(TYPE&)` is defined

- Style tip: When you write a template spell out in the documentation what assumptions you make (if any)

Basics of templates – another example

- Here is another template function example

```
template <class TYPE>
void swap(TYPE& a, TYPE& b) {
    TYPE tmp = a ; // declare generic temporary
    a = b ;
    b = tmp ;
}
```

  - Allocation of generic storage space
  - Only assumption of this swap function: `TYPE::operator=()` defined
  - Since `operator=()` has a default implementation for all types this swap function truly universal
    - Unless of course a class declares `operator=()` to be private in which case no copies can be made at all

274

Template formalities

- Formal syntax of template function
  - Template declaration

```
template <class TYPE>
TYPE function_name(TYPE& t) ;
```

  - Template definition

```
template <class TYPE>
TYPE function_name(TYPE& t){
  // body
}
```

- What's OK
  - Multiple template classes allowed

```
template <class TYPE1, class TYPE2,…class TYPEN>
TYPE1 function_name(TYPE1&, TYPE2&,… TYPEN&) ;
```

  - Non-template function arguments allowed

```
template <class TYPE>
TYPE function_name(TYPE& t, int x, float y) ;
```

275

Template formalities

- And what's not
  - Template definitions must be in the global scope

```
int myFunction() {
   template <class T>        // ERROR - not allowed
     void myTemplFunc(T& t) ;
}
```

  - Template TYPE must appear in signature

```
template <class TYPE>
TYPE function_name(int i) ; // ERROR cannot overload
                            // on return type
```

  - Reason: function overloading by return type is not allowed

Templates & files

- Template functions **cannot** be declared and defined in separate files → everything goes in the .hh file.

```
// swap.hh -- Declaration
template <class TYPE>
void swap(TYPE& a, TYPE& b) ;

template <class TYPE>
void swap(TYPE& a, TYPE& b) {
  TYPE tmp = a ;
  a = b ;
  b = tmp ;
}
```

  - Reason: When templates are instantiated, compiler must be able to see all code, including functions definitions

Template specialization

- Sometimes you have a template function that is almost generic because
  - It doesn't work (right) with certain types.
    For example `max(const char* a, const char* b)`

```
template<class TYPE>
TYPE max(const TYPE& a, const TYPE& b) {
    return (a>b) ? a : b ; // comparing pointer not sensible
}
```

  - Or for certain types there is a more efficient implementation of the algorithm

- Solution: provide a *template specialization*
  - Can only be done in definition, not in declaration
  - Tells compiler that specialized version of function for given template should be used when appropriate

```
template<>
const char* max(const char*& a, const char*& b) {
    return strcmp(a,b)>0 ? a : b ; // Use string comparison instead
}
```

278

Template classes

- Concept of templates also extends to classes
  - Can define a template class just like a template function

```
template<class T>
class Triplet {
public:
   Triplet(T& t1, T& t2, T& t3) () ;
private:
   T _array[3] ;
};
```

- Class template mechanism allows to create generic classes
  - A generic class provides the same set of behaviors for all types
  - Eliminates code duplication
  - Simplifies library design
  - Use case per excellence: container classes (arrays, stacks etc…)

279

Generic container class example

- A generic stack example

```
template<class TYPE>
class Stack {
public:
   Stack(int size) : _len(size), _top(0) {    // constructor
      _v = new TYPE[_len] ;
   }
   Stack(const Stack<TYPE>& other) ;          // copy constructor
   ~Stack() { delete[] _v ; }

   void push(const TYPE& d) { _v[_top++] = d ; }
   TYPE pop() { return _v[--_top] ; }

   Stack<TYPE>& operator=(const Stack<TYPE>& s) ; // assignment

private:
   TYPE* _v ;                    Assumptions on TYPE
   int _len ;                    -Default constructor
   int _top ;                    -Assignment defined

} ;
```


Using the generic container class

- Example using Stack

```
void example() {

   Stack<int> s(10) ; // stack of 10 integers
   Stack<String> t(20) ; // stack of 20 Strings

   s.push(1) ;
   s.push(2) ;
   cout << s.pop() << endl ;

   // OUTPUTS '2'

   t.push("Hello") ; // Exploit automatic
   t.push("World") ; // const char* → String conversion

   cout << t.pop() << " " << t.pop() << endl ;

   // OUTPUTS 'World Hello'

}
```

Initializer list of generic containers (C++ 2011)

- In C++2011 the compound initializer syntax of arrays can be extended to generic container classes

```
int x[3] = { 0, 1, 2 } ;

IntVector iv = { 0, 1, 2 } ; // Also works!

// Because constructor with initializer_list
// was added to class IntVector

class IntVector {
public:
  IntVector(std::initializer_list<int> ilist) ;
  ~IntVector() ;

private:
  int* _xvec ;
} ;
```


Initializer list of generic containers (C++ 2011)

- In C++2011 the compound initializer syntax of arrays can be extended to generic container classes
  - Retrieve content with iterator semantics – more in Module 7

```
class IntVector {
public:
  IntVector(std::initializer_list<int> ilist) {

    _xvec = new int[ilist.size()] ;

    int i(0) ;
    auto iter = ilist.begin() ;
    while (iter != ilist.end()) {
        _xvec[i++] = *iter ;
        iter++ ;
  }
  ~IntVector() ;

private:
  int* _xvec ;
} ;
```

Non-class template parameters

- You can also add non-class parameters to a template declaration, e.g.
  - parameter must be constant expression (template is instantiated at compile time!)

```
template<class T, int DIM>
class Vector {
public:
   Vector() ;
   Vector(T[] input) ;
private:
   T _array[DIM] ;
} ;
```

  - Advantage: avoid dynamic memory allocation (runtime performance penalty)

```
void example() {
  Vector<double,3> ThreeVec ;
  Vector<double,4> FourVec ;
}
```


Template default parameters

- Default values for template parameters are also allowed

```
template<class T=double, int DIM=3>
class Vector {
public:
   Vector(T[] input) ;
private:
   T _array[DIM] ;
} ;
```

```
void example() {
  Vector          ThreeVecDouble ;
  Vector<int>     ThreeVecInt ;
  Vector<float,4> FourVecFloat ;
}
```

Containment, composition

- No real limit on complexity of template constructions
  - Containment

```
template<class TYPE>
class A {
private:
  B<TYPE> member1 ;         // OK - generic template member
  C<int> member2 ;          // OK - specific template member
  D member3 ;              // OK - non-template member
public:
  A(args) : B(args),C(args),D(args) {} // initialization
} ;
```

  - Composition

```
template<class TYPE> class Vec { … } ; // Vector container
template<class TYPE> class Arr { … } ; // Array container
template<class TYPE> class Sta { … } ; // Stack container

Vec<String> c1 ;             // Vector of Strings
Vec<Arr<String> > c2 ;       // Vector of Arrays of Strings
Vec<Arr<Sta<String> > > c3 ; // Vector of Arrays of Stacks of Strings
Vec<Vec<Vec<String> > > c4 ; // Vector of Vectors of Vectors of Strings
```

Note extra space to distinguish from operator>>

286

Containment, composition

- No real limit on complexity of template constructions
  - Containment

```
template<class TYPE>
class A {
private:
  B<TYPE> member1 ;         // OK - generic template member
  C<int> member2 ;          // OK - specific template member
  D member3 ;              // OK - non-template member
public:
  A(args) : B(args),C(args),D(args) {} // initialization
} ;
```

  - Composition

```
template<class TYPE> class Vec { … } ; // Vector container
template<class TYPE> class Arr { … } ; // Array container
template<class TYPE> class Sta { … } ; // Stack container

Vec<String> c1 ;             // Vector of Strings
Vec<Arr<String>> c2 ;        // Vector of Arrays of Strings
Vec<Arr<Sta<String>>> c3 ; // Vector of Arrays of Stacks of Strings
Vec<Vec<Vec<String>>> c4 ; // Vector of Vectors of Vectors of Strings
```

No longer needed in C++2011!

287

Pointer memory management tools (C++2011)

- C++ also adds templated-based tools for pointer-based memory management
- Idea: have a dedicated wrapper class that 'owns' a pointer
  - Can be returned by-value from functions, if wrapper is deleted because it goes out of scope, it will delete the pointer
- Situation without wrapper

```
double* allocate_buffer(int size) {
    return new double[size] ;
}

int main() {
   // we own tmp, don't forget to delete
   double* tmp = allocate_buffer(100) ;
   tmp[3] = 5 ;
}
```


Pointer memory management tools (C++2011)

- C++ also adds templated-based tools for pointer-based memory management
- Idea: have a dedicated wrapper class that 'owns' a pointer
  - Can be returned by-value from functions, if wrapper is deleted because it goes out of scope, it will delete the pointer
- Situation with wrapper

```
unique_ptr<double> allocate_buffer(int size) {
    return unique_ptr<double>(new double[size]) ;
}

int main() {
   // we own tmp, don't forget to delete
   unique_ptr<double> tmp = allocate_buffer(100) ;
   tmp[3] = 5 ;
}
// memory held by tmp deleted when tmp goes out of scope
```

Pointer memory management tools (C++2011)

- C++ also adds templated-based tools for pointer-based memory management
- Idea: have a dedicated wrapper class that 'owns' a pointer
  - Can be returned by-value from functions, if wrapper is deleted because it goes out of scope, it will delete the pointer
- Situation with wrapper

```
int main() {
   // we own tmp, don't forget to delete
   unique_ptr<double> tmp = allocate_buffer(100) ;
   tmp[3] = 5 ;
}
```

Class unique_ptr overloads operator-> to return pointer to payload. Can use unique_ptr<T> in same way as T*