SFU CMPT-212 2008-1 Topic: Class Templates

Ján Maňuch

E-mail: jmanuch@sfu.ca

Monday 3rd March, 2008

Class templates

- *Question:* the class `intArray` works with `int`s — can we generalize the class `intArray` to work with other types?
- *Solution 1*: using the `typedef` keyword
  - `typedef` is used to define shortcuts for compound types:

```
1 typedef unsigned int *pointer_to_uint;
2 // pointer_to_uint is a new type name
3 pointer_to_uint a; // a is a pointer to unsigned int
```

  - instead of `int` use some abstract type, say `Item`, and `typedef Item` to desired type

  Example: arraytypedef.cpp

- *disadvantages:*
  - difficult to use
  - we can define Array only for one particular type in one program

- *Solution 2:* ("better solution") using *class templates* — basically they work in the same way as *function templates*

Example:

```
1  template <typename Item> class Array {
2    Item *data;
3    long size;
4   public:
5    Array(long s);
6    Array(const Array &);
7    virtual ~Array();
8    Array & operator=(const Array & a);
9    Item &get(long index) const;
10   void set(long index, const Item &value);
11   friend ostream & operator<<(ostream &os,
12                               const Array &a);
13 };
```

- when providing member function definitions, we have to use `template` syntax:

```
1 // definition of the method in the header file:
2 template <typename Item>
3 Array<Item>::Array(long s) {
4   size = s;
5   data = new Item[size];
6 }
```

- or

```
1 // definition of the method in the source code file
2 export template <typename Item>
3 Array<Item>::Array(long s) {
4   size = s;
5   data = new Item[size];
6 }
```

- **Remark:** Function definitions have to be: either in the same file as the *template class* declaration, or have to prefixed with the keyword `export`.

- *to use the* class template *we have to specify type* explicitly (it's not possible to deduce the type parameter from type of arguments as the are no arguments)

```
1   Array<int> a; // substitute int for Item
2   Array<double> b; // substitute double for Item
```

Example: arraytemplate.cpp

Remarks:

- There is no class `Array`, there are only classes `Array<int>` etc. Hence, in theory, you should always use `Array<Item>` and not just `Array` when declaring and defining methods of the template class.
- Inside the template class definition, `Array<Item>` is automatically substituted for `Array`

```
1  template <typename Item> class Array {
2  ...
3    Array & operator=(const Array & a); // ok
4  };
```

Remarks (continued):

- In definitions of methods of template class, the abbreviation `Array` can be used only for parameters, but not for returned types!

```
1  template <typename Item>
2  // safest:
3  Array<Item> &Array<Item>::operator=(const Array<Item> &a)
4  // ok:
5  // Array<Item> &Array<Item>::operator=(const Array &a)
6  // compiler error:
7  // Array &Array<Item>::operator=(const Array<Item> &a)
8  {
9    if (this == &a)
10   ...
11 }
```

Friends to class templates

- Friend functions to class templates are a bit tricky. A friend function can be
  - non-template friend to every instantiation of the template class (you cannot use template parameters in friend function prototype);
  - template friend to every instantiation of the template class (template parameters of the friend function are different from template parameters of the template class);
  - template friend to a particular instance (for every instantiation of the class template, we need to create a new friend function)
    — *that's what we usually want;*
    — the most complex.

Example (template friend to particular instance): (arraytemplate.cpp)

Step 1: friend function is a template function (need to be specified before template class definition):

```
1  template <typename Item> class Array;
2  // forward declaration of the class - needed for the next line
3  template <typename Item>
4  ostream & operator<<(ostream &os, const Array<Item> &a);
```

Step 2: we need to instantiate friend function in the class template:

```
1  template <typename Item> class Array {
2  ...
3    friend ostream & operator<< <Item>(ostream &os, const Array<Item> &a);
4  };
```

Step 3: provide the template definition of the friend function

```
1  template <typename Item>
2  ostream & operator<<(ostream &os, const Array<Item> &a)
3  {
4    os << "<";
5    for (long i=0; i<a.size; i++)
6      os << a.data[i] <<" ";
7    os << ">";
8    return os;
9  }
```

Example (template friend to particular instance):

OR

- define the friend function inside the template class (it becomes `inline` function, so should be used only for friend functions with short code)

```
1  template <typename Item> class Array {
2  ...
3    friend ostream & operator<<(ostream &os, const Array<Item> &a)
4    {
5      os << "<";
6      for (long i=0; i<a.size; i++)
7        os << a.data[i] <<" ";
8      os << ">";
9      return os;
10   }
11 };
```

Supplemental material:
`http://www.devx.com/cplus/10MinuteSolution/30302/1954`

Sample questions

- Write the code for a class template `SimpleArray`. It takes one type argument `T` representing the type which is stored in the array. It should represent an array of `T` of fixed size: `10`. Provide the following three functions:
  1. constructor with one parameter of type `const` reference to `T` and initializes all `10` elements of the array to the passed argument;
  2. `set` function, which takes two parameters: `int` (index) and `const` reference to `T` (a new value) and return nothing; and
  3. `get` function, which take one parameter `int` (index) and returns `const` reference to `T` (to the corresponding value).

- Write a code for a class template `Stack`. It takes one type argument `T` representing the type which is stored in the stack. Provide the constructor, destructor (freeing all dynamically allocated memory) and the methods `push`, `pop` and `isEmpty` (with arguments and return types appropriate for what they do).

Type requirements

- for each template type parameter you **should** specify all the requirements for this type
  - this is not part of the language
  - but should be included as documentation (comments)
  - important for somebody who wants to use the template (without going through the code)

Example: class template `Array`

- type parameter `Item` should have a default constructor, an assignment operator and should be printable to a stream (`ostream& operator<<(ostream&, const Item&)` should be defined)

Template parameters

- *templates* can have *more than one parameter*:

```
1  template <typename T1, typename T2>
2  class Pair {
3    T1 a;
4    T2 b;
5   public:
6    Pair(const T1 &aa, const T2 &bb) : a(aa), b(bb) {}
7  };
8  Pair<int,double> pair(2,3.1);
```

Example: pair.cpp

Note. The standard library provides template `pair` in the header file `utility`. The two elements can be accessed directly with `.first` and `.second`.

- can have *default values*:

```
1 template <typename T=double> class A {...};
2 A<> obj;
```

3 types of arguments:

- *type argument* — specified using keyword `typename` (or equivalently `class`)
- *expression argument* — specified by the type required

  Example:

```
1 template <typename T, int n>
2 class A {...};
```

  class template `A` takes to arguments, the first is a type, the second is an `int` value (a constant expression of `int` type)

```
1 A<unsigned long,10> obj;
```

  Example: arraytp.cpp [Prata]

- *template arguments* — specified using keyword `template`

  Example: Assume we have two template classes:

```
1 template <typename T> class DynArray { ... };
2 template <typename T> class LinkedList { ... };
```

  with the same interface (public methods). The only difference is that some operations are more efficient in `DynArray` (access the i-th element) and some in `LinkedList` (insert element at certain position).

  We want to create a template class `Set` which can use either (or even some other template class with the same interface) to store the elements:

```
1 template <typename Type,
2           template <typename> class Container>
3 class Set {
4 private:
5   Container<Type> elements;
6   ...
7 };
```

```
1  template <typename Type,
2            template <typename> class Container>
3  class Set {
4  private:
5    Container<Type> elements;
6    ...
7  };
```

Usage: Class template `Set` takes two arguments:

- first: a normal type (of elements in the set)
- second: a class template with 1 type argument (a container used to store elements)

hence, we can do the following:

```
1  Set<float,DynArray> objA;
2  Set<long,LinkedList> objB;
```

Example: templateparameter.cpp

Depending on the application, it might sometimes more efficient to use `DynArray` and sometimes `LinkedList`.

Template instantiations and specializations

- *implicit instantiation*

```
1 Pair<double,int> pair;
```

- *explicit instantiation*

```
1 template class Pair<double,int>;
```

- *explicit specialization*

```
1 template <> class Pair<char*,char*>
2 {
3   // provide different behavior of Pair
4   // if both arguments are char*
5 };
```

- *partial specialization:*

  Example 1:

```
1  template <typename T> class Pair<T,char*>
2  {
3    // provide different behavior of Pair
4    // if the second argument is char*
5  };
```

  Example 2:

```
1  template <typename T> class Pair<T,T>
2  {
3    // provide different behavior of Pair
4    // if the arguments are the same
5  };
```

  Example: specialization.cpp

Sample questions

- What kind of parameters class templates can have?
- Write a code for a class template `Triple`. It takes three type arguments and uses the containment method to include subobjects of these three types. It should contain a constructor (naturally, with three parameters), and methods `first()`, `second()` and `third()` returning references to corresponding subobjects. Overload the operator `<<` so that it can be used for printing objects of class template `Triple` (this is a non-template function and does not need to be a friend to the class template `Triple` as you can access all subobjects using `first()`, `second()` and `third()` methods).

  Example of usage:

```
1 Triple<long, char*,double> p(12L,"cmpt",9.87);
2 cout <<p; // prints [12,cmpt,9.87]
```

- Write the code for a class template `FixedArray`. It takes one type argument `T` representing the type which is stored in the array and one `int` argument representing the size of the array. Do not use dynamic allocations, just ordinary arrays. Provide the following methods:
  1. constructor with one parameter of type `const` reference to `T` which initializes all elements of the array to the passed argument;
  2. const and non-const versions of `operator[]` allowing access to the elements of the array;
  3. member function `print()` which will print the elements of the array.

  Solution: template.cpp

- Provide a partial specialization of the above class template in the case when `T` is a C-style string constant, i.e., a pointer to `const char` (`const char*`), which will print the values stored in the array in quotation marks (hence, only `print()` method should differ from the general version of the `FixedArray` template).

  Solution: template.cpp