Data Structures

Advances in C++ (2)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Templates**
- Self-defined header files
- The standard library **<vector>**

Items in the ADT bag

- Our ADT bag contains items.
- We implemented it as a bag of (C++) strings.
- What if we want to implement it as a bag of integers?

```
class ArrayBag : public BagInterface
{
private:
  static const int DEFAULT_CAPACITY = 6;
  string items[DEFAULT_CAPACITY];
  // ...
public:
  // ...
  bool add(const string& newEntry);
  bool remove(const string& anEntry);
  // ...
};
```

Templates

- We hope that our implementation is “general:”
  - In a bag, all items will be of a single type.
  - For different bags, the item types can be **different**.
- In C++, **templates** make this possible.
  - It can be applied on functions and classes.
  - It is not a feature of object-oriented programming.
- This is called **generic programming**.

Templates

- **C++ class templates** allows one to pass **a data-type argument** when:
  - Invoking a function defined with templates.
  - Creating an object whose class is defined with templates.
- In our example, objects of **`ArrayBag`** can be created with the actual item type passed as an argument.
  - **`ArrayBag<string> bagOfStrings;`**
  - **`ArrayBag<int> bagOfIntegers;`**
- No need to write two implementations!

Template declaration

- To declare a type parameter, use the keywords **template** and **typename**.

```
template<typename T>
class TheClassName
{
  // T can be treated as a type inside the class definition block
};
```

  - Some old codes write **class** instead of **typename**. Both are fine.
- We then do this to all member functions:

```
template<typename T>
T TheClassName<T>::f(T t)
{
  // t is a variable whose type is T
};
```

```
template<typename T>
void TheClassName<T>::f(int i)
{
  // follow the rule even if T is not used
};
```

Template invocation

- To instantiate an object, pass a type argument.

```
int main()
{
  TheClassName<int> a;
  TheClassName<double> b;
  TheClassName<AnotherClassName> c;
};
```

- The passed type will then replace all the **T**s in the class definition.

An example

- Let's start from an example with no classes.
- When we invoke **f** with **f<double>**, the function is

```
void f(double t)
{
  cout << t;
}
```

- When we invoke **f** with **f<int>**, the function is

```
void f(int t)
{
  cout << t;
}
```

That is why we see 1.

```
#include <iostream>
using namespace std;

template<typename T>
void f(T t)
{
  cout << t;
}

int main()
{
  f<double>(1.2); // 1.2
  f<int>(1.2); // 1

  return 0;
}
```

An example with two type parameters

- We may also have multiple type parameters.

```
#include <iostream>
using namespace std;

template<typename A, typename B>
void g(A a, B b)
{
  cout << a + b;
}

int main()
{
  f<double, int>(1.2, 1);
  return 0;
}
```

An example with classes

- The syntax of applying templates to classes is very similar.
  - Add the declaration line to the class definition.
  - Add the declaration line to all member function definitions.
  - For each member function definition, specify the type parameter.

```
#include <iostream>
using namespace std;

template<typename T>
class C
{
public:
  T f();
};

template<typename T>
T C<T>::f()
{
  return 1.2;
}
```

```
int main()
{
  C<int> c;
  cout << c.f() << endl;
  return 0;
}
```

Applying templates to BagInterface

- Before we use template, **BagInterface** is:

```
class BagInterface
{
public:
  virtual int getCurrentSize() const = 0;
  virtual bool isEmpty() const = 0;
  virtual bool add(const string& newEntry) = 0;
  virtual bool remove(const string& anEntry) = 0;
  virtual void clear() = 0;
  virtual int getFrequencyOf(const string& anEntry) const = 0;
  virtual bool contains(const string& anEntry) const = 0;
  virtual void print() const = 0;
};
```

Applying templates to BagInterface

- After we use template, **BagInterface** becomes:

```
template<typename ItemType> // add this line
class BagInterface
{
public:
  virtual int getCurrentSize() const = 0;
  virtual bool isEmpty() const = 0;
  virtual bool add(const ItemType& newEntry) = 0; // treat ItemType as a type
  virtual bool remove(const ItemType& anEntry) = 0;
  virtual void clear() = 0;
  virtual int getFrequencyOf(const ItemType& anEntry) const = 0;
  virtual bool contains(const ItemType& anEntry) const = 0;
  virtual void print() const = 0;
};
```

Applying templates to ArrayBag

```
template<typename ItemType>
class ArrayBag : public BagInterface<ItemType>
{
private:
  static const int DEFAULT_CAPACITY = 6;
  ItemType items[DEFAULT_CAPACITY];
  int itemCount;
  int maxItems;
  int getIndexOf(const ItemType& target) const;
public:
  // others that do not need ItemType
  bool add(const ItemType& newEntry);
  bool remove(const ItemType& anEntry);
  bool contains(const ItemType& anEntry) const;
  int getFrequencyOf(const ItemType& anEntry) const;
};
```

Applying templates to ArrayBag

```
template<typename ItemType> // add this even if not used
bool ArrayBag<ItemType>::isEmpty() const // add this even if not used
{
  return itemCount == 0;
}

template<typename ItemType>
bool ArrayBag<ItemType>::add(const ItemType& newEntry)
{ // ItemType can be used in the function definition
  bool hasRoomToAdd = (itemCount < maxItems);
  if(hasRoomToAdd)
  {
    items[itemCount] = newEntry;
    itemCount++;
  }
  return hasRoomToAdd;
}
```

Using ArrayBag with templates

- Now we can have bags for different types of items.

```
int main()
{
  ArrayBag<string> bagOfStrings;
  ArrayBag<int> bagOfIntegers;
  ArrayBag<MyClass> bagSpecial;

  bagOfStrings.add("here");
  bagOfIntegers.add(123);
  MyClass mc;
  bagSpecial(mc);

  return 0;
}
```

One final remark

- An **operation** may need a special definition for a given type.

```
template<typename ItemType>
bool ArrayBag<ItemType>::add(const ItemType& newEntry)
{
  bool hasRoomToAdd = (itemCount < maxItems);
  if(hasRoomToAdd)
  {
    items[itemCount] = newEntry;
    itemCount++;
  }
  return hasRoomToAdd;
}
```

- What if **`ItemType`** is a class with **dynamic memory allocation**?

One final remark

- What if **ItemType** is a class with **undefined comparisons**?
- We need to (re)define operations for our classes.
- We need to do **operator overloading**.
  - To be introduced in the next lecture.

```
template<typename ItemType>
int ArrayBag<ItemType>
  ::getFrequencyOf(const ItemType& anEntry) const
{
  int frequency = 0;
  int curIndex = 0;
  while(curIndex < itemCount)
  {
    if(items[curIndex] == anEntry)
      frequency++;
    curIndex++;
  }
  return frequency;
}
```

```
class RaNum
{
private:
  int num;
  int deno;
public:
  // ...
};
```

Outline

- Templates
- **Self-defined header files**
- The standard library **<vector>**

Libraries

- There are many C++ standard **libraries**.
  - **<iostream>**, **<climits>**, **<cmath>**, **<cctype>**, **<cstring>**, etc.
- We may also want to define **our own libraries**.
  - Especially when we collaborate with others.
  - Typically, one implements classes or global functions for the others to use.
  - That function can be defined in a self-defined library.
- A library includes a **header file** (.h) and a **source file** (.cpp).
  - The header file contains declarations
  - The source file contains definitions.

Example

- Consider the following program with a single function **myMax()**:

```
#include <iostream>
using namespace std;

int myMax(int [], int);
int main()
{
  int a[5] = {7, 2, 5, 8, 9};
  cout << myMax(a, 5);
  return 0;
}
```

```
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
```

- Let's define a constant **variable** for the array length in **a header file**.

Defining variables in a library

myMax.h

```
const int LEN = 5;
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int myMax(int [], int);
int main()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  cout << myMax (a, LEN);
  return 0;
}
```

```
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
```

Including a header file

- When your main program wants to include a self-defined header file, simply indicate its path and file name.
  - `#include "myMax.h"`
  - `#include "D:/test/myMax.h"`
  - `#include "lib/myMax.h"`
  - Using `\` or `/` does not matter (on Windows).
- We still compile the main program as usual.
- Let's also define **functions** in our library!
  - Now we need a source file.

Defining functions in a library

myMax.h

```
const int LEN = 5;
int myMax(int [], int);
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int main()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  cout << myMax(a, LEN);
  return 0;
}
```

myMax.cpp

```
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
```

Including a header and a source file

- When your main program also wants to include a self-defined source file, the include statement needs not be changed.
  - `#include "myMax.h"`
- We add a source file myMax.cpp.
  - In the source file, we **implement** those functions declared in the header file.
  - The main file names of the header and source files can be different.
- The two source files (main.cpp and myMax.cpp) must be **compiled together**.
  - Each environment has its own way.

Defining one more function

myMax.h

```
const int LEN = 5;
int myMax (int [], int);
void print(int);
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int main()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  print(myMax(a, LEN));
  return 0;
}
```

myMax.cpp

```
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
void print(int i)
{
  cout << i; // cout undefined!
}
```

Defining one more function

- Each source file contains statements to run.
- Each source file must include the libraries it needs for its statements.

```
#include <iostream>
using namespace std;
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
void print(int i)
{
  cout << i; // good!
}
```

The complete set of files

myMax.h

```
const int LEN = 5;
int myMax(int [], int);
void print(int);
```

main.cpp

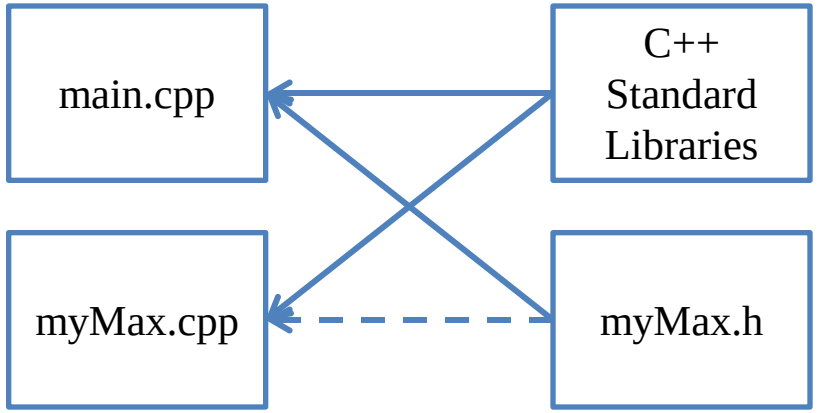
```
#include <iostream>
#include "myMax.h"
using namespace std;

int main()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  print(myMax (a, LEN));
  return 0;
}
```

myMax.cpp

```
#include <iostream>
using namespace std;
int myMax(int a[], int len)
{
  int max = a[0];
  for(int i = 1; i < len; i++)
  {
    if(a[i] > max)
      max = a[i];
  }
  return max;
}
void print(int i)
{
  cout << i;
}
```

Remarks

- In many cases, myMax.cpp also include myMax.h.
  - E.g., if `LEN` is accessed in myMax.cpp.
- More will be discussed in further courses (e.g., Data Structures).
  - More than two source files.
  - A header file including another header file.



Defining header files for `ArrayBag`

- First, we put the definition of **`BagInterface`** into a file BagInterface.h.
- In general, the inclusion relationship can be very complicated.
  - One thing may be defined **multiple** times (which is an error) if one file is included multiple times.
- To avoid this,
  - We use **`#define`** to define a token.
  - **`#ifndef`** checks whether a given token has been defined;
  - **`#endif`** labels the end of a block.

```
#ifndef _BAG_INTERFACE
#define _BAG_INTERFACE

template<typename ItemType>
class BagInterface
{
  // all those member functions
};
#endif
```

(BagInterface.h)

Defining header files for ArrayBag

- Second, we put the definition of **ArrayBag** into ArrayBag.h.
- We still use **#define**, **#ifndef**, and **#endif** to avoid multiple definitions.
- Because this class needs an implementation, we add **an include statement** at the end to include the implementation file ArrayBag.cpp.

```
#ifndef _ARRAY_BAG
#define _ARRAY_BAG
#include "BagInterface.h"

template<typename ItemType>
class ArrayBag : public BagInterface<ItemType>
{
  // all those member variables and functions
};
#include "ArrayBag.cpp"
#endif
```

(ArrayBag.h)

Defining header files for `ArrayBag`

- Third, we put the implementation of **`ArrayBag`** into ArrayBag.cpp.
- In ArrayBag.cpp, we need to **include the header file** ArrayBag.h.
- When a **standard library** is used in an implementation file, we need to include it.

```
#include <iostream>
#include "ArrayBag.h"
using namespace std;

// all other function definitions
template<typename ItemType>
void ArrayBag<ItemType>::print() const
{
  for(int i = 0; i < itemCount; i++)
    cout << items[i] << " ";
  cout << endl;
}
// all other function definitions
```

(ArrayBag.cpp)

Defining header files for `ArrayBag`

- Now, for any client who uses the class **`ArrayBag`**, all it needs to do is to **include ArrayBag.h**.
- For those who want to understand how **`ArrayBag`** should be used, they only need to read the header file.
  - **Comments** in ArrayBag.h should be understandable to client developers.
  - Implementation details should not be disclosed.

```
#include <iostream>
#include <string>
#include "ArrayBag.h"
using namespace std;

int main()
{
  ArrayBag<string> bag;

  return 0;
}
```

(main.cpp)

Outline

- Templates
- Self-defined header files
- **The standard library <vector>**

Displaying bag items

- In the textbook, the implementation of **ArrayBag** does not contain **print()**.
- To allow one to display bag items, a member function **toVector()** and a global function **displayBag()** are defined:

```
template<class ItemType>
vector<ItemType> ArrayBag<ItemType>
  ::toVector() const
{
  vector<ItemType> bagContents;
  for(int i = 0; i < itemCount; i++)
    bagContents.push_back(items[i]);
  return bagContents;
}
```

```
void displayBag(ArrayBag<string>& bag)
{
  cout << "The bag contains "
       << bag.getCurrentSize()
       << " items:" << endl;
  vector<string> bagItems = bag.toVector();
  int numberOfEntries = (int) bagItems.size();
  for (int i = 0; i < numberOfEntries; i++)
    cout << bagItems[i] << " ";
  cout << endl << endl;
}
```

The standard library <vector>

- **The class vector** with templates is defined and implemented in the standard library **<vector>**.
- It is just a "**dynamic vector**" of any type.
  - It is a class with an embedded one-dimensional dynamic array.
  - It has many useful member functions (including overloaded operators).
  - It is implemented with templates.
- Four member functions are used in the textbook implementation:
  - Constructor: `vector<ItemType> bagContents;`
  - Element addition: `bagContents.push_back(items[i]);`
  - Assignment operator: `vector<string> bagItems = bag.toVector();`
  - Indexing operator: `cout << bagItems[i] << " ";`

Some questions regarding `vector`

- Why not just **`print()`**?
  - Using **`toVector()`** rather than **`print()`** completely **decouples** **`ArrayBag`** operations and program I/O (e.g., displaying bag items on a screen).
- Should we have **`toVector()`** or **`print()`**?
  - If it is only used during **the development phase**, we should remove it after **`ArrayBag`** is completed.
- If we have **`vector`**, why should we implement **`ArrayBag`**?
- (Personal suggestion) **Try not to use `vector`** unless you are able to implement something with the same capability.