

CS1102C Data Structures and Algorithms
Semester 2 2008/2009

Week 2nd February of 2009
Tutorial 2 Suggested Solution
More on C++

Section A. Questions to be discussed in tutorial

1. (Simple Class) There are a number of omissions in the **BankAcct** class in lecture 2 due to the slide limitation. Discuss whether the following functionality is required. If so, draft the necessary implementation:
 - a. Let the user to read the **_acctNum** and **_balance** attributes
 - b. Let the user to write the **_acctNum** and **_balance** attributes
 - c. Write a printing method to show the information of bank account. Do we need (a) and (c) at the same time?
 - d. Do we need destructor for **BankAcct** class?

ANS:

- a. Yes, these access methods (known as **accessor / getter**) are useful.

```
class BankAcct {  
  
public:  
    int getAccountNumber( ) {  
        return _acctNum;  
    }  
    // similar idea for getBalance  
};
```

- b. These writing methods (known as **mutator / setter**) should be provided after careful consideration. Providing both (a) and (b) basically make the attribute public, which violate the encapsulation principle. In this example, it is better **not to provide** these mutator methods to avoid misuse by the users.
- c. If (a) is provided, this may not be necessary, as the user can access each attribute and print them out if needed. However, providing the printing method can make the user life easier. It is recommended that you always provide a print() method for any class you write. The method can come in useful during debugging and development.

There are two ways to provide the printing method: a directly print the information to standard output OR return the information returned as a string. The latter is more flexible as it allows the user to decide what to do with that information. Both versions are shown below:

```

class BankAcct {
public:
    void print( ) {
        cout << "Account Number: " << _acctNum << endl;;
        ... //similar idea for other attributes
    }

    string toString(){
        //make use of string output string
        ostream os;

        os << "Account Number: " << _acctNum << endl;
        ...

        return os.str();
    }
};

```

d. The **BankAcct** object stores only simple data, so there is no need for any special cleanup code. So, destructor is not needed in this case.

2. (Template Function) Below is a very useful function to find the largest integer in an integer array:

```

int findMaxIdx( int array[], int size)
//purpose: Locate the largest item in array[size]
//return: Return the index of the largest item
//assumption: the array contains at least one item
{
    int maxIdx = 0, i;

    for (i = 1; i < size; i++) {
        if (array[i] > array[maxIdx])
            maxIdx = i;
    }
    return maxIdx;
}

```

Show how to rewrite the above as a **template function**.

ANS:

```

template<typename T>
int findMaxIdx( T array[], int size)
//purpose: Locate the largest item in array[size]
//return: Return the index of the largest item
//assumption: the array contains at least one item
{
    int maxIdx = 0, i;

    for (i = 1; i < size; i++) {
        if (array[i] > array[maxIdx])
            maxIdx = i;
    }
    return maxIdx;
}

```

```
}
```

It is incorrect to change the return type and the datatype for `size` to `T`. (why?)

3. (Input) Suppose we need a number of **integer** values from a user. Show the possible C++ code if the input is formatted as follows. Remember to store **all the values** as well as the number of values received in each case.
- The values are entered as a single line, with space(s) in between.
e.g. 1 2 3 5 6 101 [enter]
 - The values are entered one at a time, until the special value -9999 is entered.
e.g. 1
2
....
-9999
 - The values are entered one at a time, until the user press “Ctrl-D”. This special keypress represent the “end of file”. This is the same as what your program will receive when the end of an input file is reached.
e.g. 1
2
....
[Ctrl-D]

ANS:

a.

```
string input;
vector<int> values;

getline(cin, str);    //get the whole line

istringstream iss( str );    //make use of string input stream for reading

int singleValue;

while (iss >> singleValue){
    values.push_back(singleValue);
}
// Number of values = values.size();
// The solution can use an array instead. However, you will need to make sure the
// array is big enough to store all the values. Vector is the best choice in this case.
```

b.

```
int singleValue;
vector<int> values;

cin >> singleValue;
while (singleValue != -9999){
    values.push_back(singleValue);
    cin >> singleValue;
}
```

c.

```
int singleValue;  
vector<int> values;  
  
while ( cin >> singleValue ) {  
    values.push_back(singleValue);  
}
```