14.1 Function templates

Multiple functions may be nearly identical, differing only in their data types, as below.

Figure 14.1.1: Functions may have identical behavior, differing only in data types.


```
// Find the minimum of three **ints**
int TripleMinInt(int item1, int item2, int item3) {
   int minVal;

   minVal = item1;

   if (item2 < minVal) {
      minVal = item2;
   }
   if (item3 < minVal) {
      minVal = item3;
   }
   return minVal;
}
```

```
// Find the minimum of three **chars**
char TripleMinChar(char item1, char item2, char item3) {
   char minVal;

   minVal = item1;

   if (item2 < minVal) {
      minVal = item2;
   }
   if (item3 < minVal) {
      minVal = item3;
   }
   return minVal;
}
```

Writing and maintaining redundant functions that only differ by data type can be time-consuming and error-prone. The language supports a better approach.

A ***function template*** is a function definition having a special type parameter that may be used in place of types in the function.

Figure 14.1.2: A function template enables a function to handle various data types.

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

template<typename TheType>
TheType TripleMin(TheType item1, TheType item2, TheType item3) {
   TheType minVal = item1; // Holds min item value, init to first item

   if (item2 < minVal) {
      minVal = item2;
   }
   if (item3 < minVal) {
      minVal = item3;
   }

   return minVal;
}

int main() {
   int num1 = 55;        // Test case 1, item1
   int num2 = 99;        // Test case 1, item2
   int num3 = 66;        // Test case 1, item3

   char let1 = 'a';      // Test case 2, item1
   char let2 = 'z';      // Test case 2, item2
   char let3 = 'm';      // Test case 2, item3

   string str1 = "zzz";     // Test case 3, item1
   string str2 = "aaa";     // Test case 3, item2
   string str3 = "mmm";     // Test case 3, item3

   // Try TripleMin function with ints
   cout << "Items: " << num1 << " " << num2 << " " << num3 << endl;
   cout << "Min: " << TripleMin(num1, num2, num3) << endl << endl;

   // Try TripleMin function with chars
   cout << "Items: " << let1 << " " << let2 << " " << let3 << endl;
   cout << "Min: " << TripleMin(let1, let2, let3) << endl << endl;

   // Try TripleMin function with strings
   cout << "Items: " << str1 << " " << str2 << " " << str3 << endl;
   cout << "Min: " << TripleMin(str1, str2, str3) << endl << endl;

   return 0;
}
```



```
Items: 55 99 66
Min: 55

Items: a z m
Min: a

Items: zzz aaa mmm
Min: aaa
```

The function return type is preceded by `template<typename TheType>` (highlighted yellow), where TheType can be any identifier. That type is known as a ***type parameter*** and can be used throughout the function for any parameter types, return types, or local variable types (highlighted orange). The identifier is known as a template parameter, and may be various reference types or even another template parameter.

The compiler automatically generates a unique function definition for each type appearing in function calls to the function template. Thus, the above example's calls would create three TripleMin() function

definitions using int, char, and string as in this section's introductory example. The programmer never sees those function definitions.

PARTICIPATION ACTIVITY 14.1.1: Function templates.

1) Fill in the blank.

```
template<typename MyType>
_____ MyAvg3 (MyType i, MyType j,
MyType k) {
   return (i + j + k)/3;
}
```

- TheType
- int
- MyType

2) Fill in the blank.

```
template<typename _________>
T TripleMin(T item1, T item2, T
item3) { ... }
```

- int
- TheType
- T
- Not possible; T is not a valid type.

3) For the earlier TripleMin function template, what happens if a call is TripleMin(i, j, k) but those arguments are of type long long?

- The compiler generates an error message because only int and char are supported.
- During runtime, the long long values are forced to be int values.
- The compiler creates function with long long types and calls that function.

4) For the earlier TripleMin function template, what happens if a call is TripleMin(i, j, k) but those arguments are of type string?

- string is just another type, so the function will compare strings.

- The compiler generates an error, because only numerical types can be passed.

5) For the earlier TripleMin function template, what happens if a call is TripleMin(i, j, z), where i and j are ints, but z is a string?

- The function will compare the ints and the string.
- The compiler will generate an error, because TheType must be the same for all three arguments.

Programmers optionally may explicitly specify the type as a special argument, as in `TripleMin<int>(num1, num2, num3);`.

A function template may have multiple parameters:

Construct 14.1.1: Function template with multiple parameters.

```
template<typename T1, typename T2>
ReturnType FunctionName(Parameters) {
   ...
}
```

Earlier versions of C++ used the word "class" rather than "typename". Though misleading (the type need not be a class, but can be int or double, for example), the word class is still allowed for backwards compatibility, and much existing C++ code uses that word.

zyDE 14.1.1: Function templates with multiple parameters.

This program currently fails to compile. Modify TripleMin() so that item1 can be of a diff type than item2 and item3.

Load default template... Run

```
1
2 #include <iostream>
3 using namespace std;
4
5 template<typename TheType>
```

```
TheType TripleMin(TheType item1, TheTyp
   TheType minVal = item1; // Holds mir

   if (item2 < minVal) {
      minVal = item2;
   }
   if (item3 < minVal) {
      minVal = item3;
   }

   return minVal;
}
```


Exploring further:

- Templates from cplusplus.com
- Function templates from msdn.microsoft.com

14.2 Class templates

Multiple classes may be nearly identical, differing only in their data types. The following shows a class managing three int numbers, and a nearly identical class managing three short numbers.

Figure 14.2.1: Classes may be nearly identical, differing only in data type.

```
class TripleInt {
public:
   TripleInt(int val1 = 0, int val2 = 0, int val3 = 0);
   void PrintAll() const; // Print all data member values
   int MinItem() const;   // Return min data member value
private:
   int item1;             // Data value 1
   int item2;             // Data value 2
   int item3;             // Data value 3
};

TripleInt::TripleInt(int i1, int i2, int i3) {
   item1 = i1;
   item2 = i2;
   item3 = i3;
}

// Print all data member values
void TripleInt::PrintAll() const {
   cout << "(" << item1 << "," << item2
        << "," << item3 << ")" << endl;
}

// Return min data member value
int TripleInt::MinItem() const {
   int minVal;

   minVal = item1; // Holds min item value, init to first item

   if (item2 < minVal) {
      minVal = item2;
   }
   if (item3 < minVal) {
      minVal = item3;
   }

   return minVal;
}
```


©zyBooks 04/25/21 07:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

```
class TripleShort {
public:
   TripleShort(short val1 = 0, short val2 = 0, short val3 = 0);
   void PrintAll() const; // Print all data member values
   short MinItem() const; // Return min data member value
private:
   short item1;           // Data value 1
   short item2;           // Data value 2
   short item3;           // Data value 3
};

TripleShort::TripleShort(short i1, short i2, short i3) {
   item1 = i1;
   item2 = i2;
   item3 = i3;
}

// Print all data member values
void TripleShort::PrintAll() const {
   cout << "(" << item1 << "," << item2
        << "," << item3 << ")" << endl;
}

// Return min data member value
short TripleShort::MinItem() const {
   short minVal;

   minVal = item1; // Holds min item value, init to first item

   if (item2 < minVal) {
      minVal = item2;
   }
   if (item3 < minVal) {
      minVal = item3;
   }

   return minVal;
}
```


Writing and maintaining redundant classes that only differ by data type can be time-consuming and error-prone. The language supports a better approach.

A ***class template*** is a class definition having a special type parameter that may be used in place of types in the class. A variable declared of that class type must indicate a specific type.

Figure 14.2.2: A class template enables one class to handle various data types.

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

template<typename TheType>
class TripleItem {
public:
   TripleItem(TheType val1 = 0, TheType val2 = 0, TheType val3 = 0);
   void PrintAll() const;   // Print all data member values
   TheType MinItem() const; // Return min data member value
private:
   TheType item1;           // Data value 1
   TheType item2;           // Data value 2
   TheType item3;           // Data value 3
};

template<typename TheType>
TripleItem<TheType>::TripleItem(TheType i1, TheType i2, TheType i3) {
   item1 = i1;
   item2 = i2;
   item3 = i3;
}

// Print all data member values
template<typename TheType>
void TripleItem<TheType>::PrintAll() const {
   cout << "(" << item1 << "," << item2
        << "," << item3 << ")" << endl;
}

// Return min data member value
template<typename TheType>
TheType TripleItem<TheType>::MinItem() const {
   TheType minVal = item1; // Holds value of min item, init to first item

   if (item2 < minVal) {
      minVal = item2;
   }
   if (item3 < minVal) {
      minVal = item3;
   }

   return minVal;
}

int main() {
   TripleItem<int> triInts(9999, 5555, 6666); // TripleItem class with ints
   TripleItem<short> triShorts(99, 55, 66);   // TripleItem class with shorts

   // Try functions from TripleItem
   triInts.PrintAll();
   cout << "Min: " << triInts.MinItem() << endl << endl;

   triShorts.PrintAll();
   cout << "Min: " << triShorts.MinItem() << endl << endl;

   return 0;
}
```


©zyBooks 04/25/21 07:43 488201
xiang zhao
BAYLORCSI14301440Spring2021

```
(9999,5555,6666)
Min: 5555

(99,55,66)
Min: 55
```

The class declaration is preceded by `template<typename TheType>` (highlighted yellow), where TheType can be any identifier. That type is known as a ***template parameter*** and can be used throughout the class, such as for parameter types, function return types, or local variable types. The identifier is known as a template parameter, and may be various items such as an int, double, char, or string, or a pointer or reference, or even another template parameter.

Any of the class's functions defined outside the class declaration must also be preceded by the template declaration, and have the type in angle brackets appended to its name as in `void TripleItem<TheType>::Print()`. An object of this class can be declared by appending after the class name a specific type in angle brackets (highlighted orange), such as `TripleItem<short> triShorts(99,55,66);`.

PARTICIPATION ACTIVITY 14.2.1: Class templates.

1) A class has been defined using the type GenType throughout, where GenType is intended to be chosen by the programmer when declaring a variable of this class. The code that should immediately precede the class definition is `template<GenType>`
   - ○ True
   - ○ False

2) A key advantage of a class template is relieving the programmer from having to write redundant code that differs only by type.
   - ○ True
   - ○ False

3) For a class template defined as `template<typename T> class Vehicle { ... }` an appropriate variable declaration of that class would be `Vehicle<T> v1;`
   - ○ True
   - ○ False

A template may have multiple parameters, separated by commas:

Construct 14.2.1: Class template with multiple parameters.

```
template<typename T1, typename T2>
class ClassName {
   ...
};
```


Earlier versions of C++ used the word "class" rather than "typename". Though misleading (the type need not be a class, but can be int or double, for example), the word class is still allowed for backwards compatibility, and much existing C++ code uses that word.

Note that C++'s vector class is a class template, which is why a variable declared as a vector indicates the type in angle brackets, as in `vector<int> nums(100);`.

zyDE 14.2.1: Class templates.

Modify the TimeHrMn class to utilize a class template. Note that the main() function pa
int and double as parameters for the SetTime() member function.

Load default template... Run

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

class TimeHrMn {
public:
   void SetTime(int userMin);
   void PrintTime() const;
private:
   int hrsVal;
   int minsVal;
};

void TimeHrMn::SetTime(int userMin) {
   minsVal = userMin;
   hrsVal = userMin / 60.0;
}
```


Exploring further:

- Templates from cplusplus.com

14.3 C++ example: Map values using a function template

zyDE 14.3.1: Map a value using a function template.

The program below uses a function template to map numeric, string, or character value shorter list of values. The program demonstrates a mapping for integers using a table o

```
100
200
300
400
500
600
```

The program gets an integer value from a user and returns the first value in the table tha greater than or equal to the user value, or the user value itself if that value is greater thar largest value in the table. Ex:

```
165 returns 200
444 returns 500
888 returns 888
```

1. Run the program and notice the input value 137 is mapped to 200. Try changing th input value and running again.
2. Modify the program to call the GetMapping function for a double and a string, sim the integer.
3. Run the program again and enter an integer, a double, and a string

Load default templ

```
1 #include <iostream>
2 #include <vector>
3 #include <string>
4 using namespace std;
5
6 template<typename MapType>
7 MapType GetMapping(MapType mapMe, vector<MapType> mappings){
8    MapType result = mapMe;
```

```
   unsigned int i;
   bool keepLooking;

   keepLooking = true;

   cout << endl;
   cout << "Mapping range: ";
   for (i = 0; i < mappings.size(); ++i) {
      cout << mappings.at(i) << " ";
```



```
137
```

Run

zyDE 14.3.2: Map a value using a function template (solution).

A solution to the above problem follows.

Load default templ

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

template<typename MapType>
MapType GetMapping(MapType mapMe, vector<MapType> mappings){
   MapType result = mapMe;
   unsigned int i;
   bool keepLooking;

   keepLooking = true;

   cout << endl;
   cout << "Mapping range: ";
   for (i = 0; i < mappings.size(); ++i) {
      cout << mappings.at(i) << " ";
```



```
137
4.44444
Hi
```

Run

14.4 Practice Practicum

This section's content is not available for print.

14.5 Practice Practicum

This section's content is not available for print.

14.8 LAB: What order? (function templates)

Define a generic function called CheckOrder() that checks if four items are in ascending, neither, or descending order. The function should return -1 if the items are in ascending order, 0 if the items are unordered, and 1 if the items are in descending order.

The program reads four items from input and outputs if the items are ordered. The items can be different types, including integers, strings, characters, or doubles.

Ex. If the input is:

```
bat hat mat sat
63.2 96.5 100.1 123.5
```


the output is:

```
Order: -1
Order: -1
```

LAB ACTIVITY 14.8.1: LAB: What order? (function templates) 0 / 10

main.cpp Load default template...

```
#include <string>
#include <iostream>

using namespace std;

// TODO: Define a generic method called CheckOrder() that
//       takes in four variables of generic type as arguments.
//       The return type of the method is integer

   // Check the order of the input: return -1 for ascending,
   // 0 for neither, 1 for descending


int main(int argc, char* argv[]) {
   // Read in four strings
   string stringArg1, stringArg2, stringArg3, stringArg4;
   cin >> stringArg1;
   cin >> stringArg2;
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.cpp (Your program) → Outp

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

14.9 LAB: Zip code and population (class templates)

Define a class `StatePair` with two template types (`T1` and `T2`), a constructor, mutators, accessors, and a PrintInfo() method. Three vectors have been pre-filled with StatePair data in main():

- `vector<StatePair <int, string>> zipCodeState`: ZIP code - state abbreviation pairs
- `vector<StatePair<string, string>> abbrevState`: state abbreviation - state name pairs
- `vector<StatePair<string, int>> statePopulation`: state name - population pairs

Complete main() to use an input ZIP code to retrieve the correct state abbreviation from the vector zipCodeState. Then use the state abbreviation to retrieve the state name from the vector abbrevState. Lastly, use the state name to retrieve the correct state name/population pair from the vector statePopulation and output the pair.

Ex: If the input is:

```
21044
```

the output is:

```
Maryland: 6079602
```

LAB ACTIVITY | 14.9.1: LAB: Zip code and population (class templates) | 0 / 10

Current file: **main.cpp** Load default template...

```
1  #include<iostream>
2  #include <fstream>
3  #include <vector>
4  #include <string>
5  #include "StatePair.h"
6  using namespace std;
7
8  int main() {
9     ifstream inFS; // File input stream
10    int zip;
11    int population;
12    string abbrev;
13    string state;
14    unsigned int i;
15
16    // ZIP code - state abbrev. pairs
17    vector<StatePair <int, string>> zipCodeState;
```

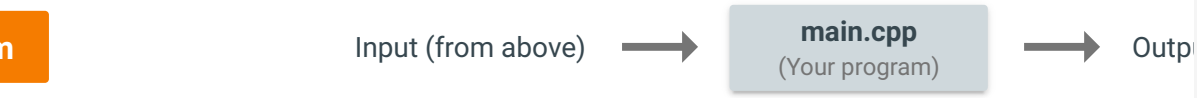

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

14.10 Lab 6 - Class Templates

CSI 1440

Lab 6

“Class Templates”

Templates

Templates in C++ is not a difficult idea. This lab is not intended to help you understand the details of how templates work. It is only intented to give students an opportunity to start developing code using templated classes.

Everyone in the class has had to come to grips with the usage of variables in general. With templates, you can think of the type of the variable being a variable itself. The programmer is just saying that when you want to instantiate an instance of this class you will need to tell the compiler which variable type the class needs to use. After understanding that, templates just become a syntactical change.

Developing a templated class is not difficult either. I think the syntax is the most difficult aspect. It's just pure ugly. So, I suggest that you approach the MyArray class as if it was the IntArray class. Yes, I'm suggesting that you write pushBack, popBack, getSize, isEmpty, operator[], and toString as if you were writing the IntArray class we developed in the Big 4 lab. After completing that implementation and testing for correctness, go back and change all necessary instances of int to the template variable.

Here's the MyArray class written as MyInt from the Big 4 lab.

```
// You won't be able to inherit from ContainerIfc at this point which
means you may
// have to test locally to begin with.
class MyArray  {
  private:
      int *data;
      int capacity, size;

  public:
      // You will need to write all function defintions outside of the
class defintion
      MyArray();
};

MyArray::MyArray(){
   this->capacity = 5;
   this->size = 0;
   this->data = new int[this->capacity];
}
```


Here's the MyArray class with the inheritance and templating in place.

```
template <class T>
class MyArray : public ContainerIfc<T> {
  private:
      T *data;
      int capacity, size;

  public:
      MyArray();
};

// notice the T used in the name of the class not the name of the
function
template <class T>
MyArray<T>::MyArray(){
   this->capacity = 5;
   this->size = 0;
   this->data = new T[this->capacity];
}
```


MyArray Class

MyArray will inherit from ContainerIfc. ContainerIfc is an abstract classes which requires you to implement the functions in your MyArray. In addition to overriding the pure virtual functions, you will

need to implement the big four because of the precense of dynamic memory in the class.

ContainerIfc.h

```
#include <string>

using namespade std;

class BADINDEX{};

template <class T>
class ContainerIfc {
public:
  virtual ContainerIfc<T>& pushBack(T) = 0;
  virtual ContainerIfc<T>& popBack(T &) = 0;  // throws BADINDEX
  virtual int getSize() = 0;
  virtual bool isEmpty() = 0;
  virtual T& operator[](int) = 0;  // throws BADINDEX
  virtual string toString() = 0;
};
```


Memory Requirements

You are required to follow a familiar memory scheme. Your MyArray should start with enough space to store 5 base objects. It should increase by 5 base objects when needed. It should never use less than 1/4 of the capacity which should be corrected by decreasing by 5 if needed. And, it should never drop below 5 allocated integers.

toString

You are required to implement a toString function for your MyArray. toString is a function commonly used to pass classes around in a simple format. It is commonly used for logging or debugging. For example, you can print the current state of your object using the toString method before and after a suspicious line of code to see changes made. Your toString function should print the object out using the following format:

```
[size]
[capacity]
[val1 val2 val3 ….]
```


toString example

```
6
10
1 22 4 54 9 45
```

Driver.cpp

I'm supplying a driver within the zyBooks lab, but you will most likely want one for testing locally.

LAB ACTIVITY 14.10.1: Lab 6 - Class Templates 0 / 1

Current file: **main.cpp** Load default template...


```
#include <iostream>
#include "MyArray.h"

using namespace std;

template <class T>
void copyIt(MyArray<T> b) {
   cout << b.toString() << endl;
}

int main() {
   MyArray<int> a;
   int val;

   cout << "Testing default constructor..." << endl;
   cout << "Size: " << a.getSize() << " should be 0" << endl;

   cout << endl << "Testing toString..." << endl;
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.cpp** (Your program) → Outp

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

14.11 Lab 7 - Templates, Binary Files, and Exceptions

CSI 1440

Lab 7

"Templates, Binary Files, and Exceptions"

This week we are going to become more acquainted with binary files with objects.

Deliverables

- lab7-MyArray.h

MyArray Class

MyArray will inherit from ContainerIfc which inherits from SearializableIfc. Both are abstract classes which require you to implement the functions in your MyArray. In addition to overriding the pure virtual functions, you will need to implement the big four.

lab7-SerializableIfc.h

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

class SerializableIfc {
public:
  virtual void writeObject( ostream & ) = 0;
  virtual void readObject( istream & ) = 0;
};
```

lab7-ContainerIfc.h


```
#include "lab7-SerializableIfc.h"
#include <string>

class BadIndex{};

template <class T>
class ContainerIfc : public SerializableIfc {
public:
```

```
  virtual ContainerIfc<T>& pushBack(T) = 0;
  virtual ContainerIfc<T>& popBack(T &) = 0;
  virtual int getSize() = 0;
  virtual bool isEmpty() = 0;
  virtual T& operator[](int) = 0;
  virtual string toString() = 0;
};
```


Memory Requirements

You are required to follow a familiar memory scheme. Your MyArray should start with enough space to store 5 base objects. It should increase by 5 base objects when needed. It should never use less than 1/4 of the capacity which should be corrected by decreasing by 5 if needed. And, it should never drop below 5 allocated integers.

toString

You are required to implement a toString function for your MyArray. toString is a function commonly used to pass classes around in a simple format. It is commonly used for logging or debugging. For example, you can print the current state of your object using the toString method before and after a suspicious line of code to see changes made. Your toString function should print the object out using the following format:

```
[size]
[capacity]
[val1 val2 val3 ….]
```

toString sample

```
6
10
1 22 4 54 9 45
```

Binary File Format

The following file format should be assumed for reading and writing an MyArray.

```
[size - 4 byte int][capacity - 4 byte int][size bytes of data * size of
data]
```


lab7-main.cpp

I have provided a test driver for your MyArray implementation. Don't forget, for labs, that you can download your solution locally and test your code using your IDE. Also, you can make changes to the provided driver to increase testing of your class. Just realize that the output will not match if you do so.

LAB ACTIVITY | 14.11.1: Lab 7 - Templates, Binary Files, and Exceptions | 0 / 1

Current file: lab7-main.cpp ▾ Load default template...

```
1  #include <iostream>
2  #include <fstream>
3  #include "lab7-MyArray.h"
4
5  using namespace std;
6
7  template <class T>
8  void copyIt(MyArray<T> b) {
9     cout << b.toString() << endl;
10 }
11
12 int main() {
13    MyArray<int> a;
14    int val;
15
16    cout << "Testing default constructor..." << endl;
17    cout << "Size: " << a.getSize() << " should be 0" << endl;
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → lab7-main.cpp (Your program) → Out

Program output displayed here

Signature of your work What is this?

History of your effort will appear here once you begin working on this zyLab.