Data Structures and Objects
CSIS 3700

Lab 5 — Generic Containers

Goal

Experiment with various methods for creating containers that can hold multiple types of data within the same program.

Method 1 — Unions

So far, you have seen a **class** — a collection of data and the code that manipulates the data. You may have also seen a **struct**, which is just the collection of data without the code.

A **union** is similar to a **struct**. The difference is, while a **struct** can hold many values inside of itself simultaneously, a **union** can only hold one value at a time. Consider the following **struct**:

```
1 struct {
2   int foo;
3   double bar;
4 };
```

This **struct** can hold an integer **foo** value and a double-precision value **bar** simultaneously.

Now, consider the following **union**:

```
1 union {
2   int foo;
3   double bar;
4 };
```

This can hold either **foo** or **bar**, but not both simultaneously. The **struct** has enough space to hold both — 12 bytes on most systems — while the **union** only has enough space for the largest of its items — the **double**, which is 8 bytes.

To try this method out, you will need to modify the following files:

- `simpleStack.h`
- `simpleStack.cc`
- `stackTest1.cc`

First, modify `simpleStack.h`, replacing the **typedef** line with the following:

```
1 union UItem {
2   int iVal;
3   double dVal;
4 };
5
6 typedef union UItem StackType;
```

This defines the **union** and creates an alias **StackType**; the compiler will replace all occurrences of **StackType** with **union UItem**.

In `stackTest1.cc`, create a single **StackType** variable; add

```
1 StackType digit;
```

as a local variable inside **main()**. Replace the **push()** call with

```
1 digit.iVal = d;
2 myStack.push(digit);
```

Finally, replace the **pop()** and **cout** lines with

```
1 digit = myStack.pop();
2 cout << digit.iVal << endl;
```

Compile `stackTest1.cc` and `simpleStack.cc` with one `g++` command; no `Makefile` is necessary here. Test the program; it should read a number and output the digits one at a time in the proper order.

Method 2 — Zen containers

"Argue for your limitations, and sure enough, they're yours." — *Messiah's Handbook*, from Richard Bach's *Illusions*

Another method for containing multiple data types is to create a container that stores no data type. Some containers, such as stacks, queues and linear lists, do not place any special restrictions on the type of data stored within them. In fact, these containers completely ignore the data type, other than to use it to determine how many bytes to move into or out of the container. If that is the case, then it is possible to devise a container where no type is specified; only the number of bytes to move is given.

I call this type of approach a "zen" data structure, as it has a distinctly zen flavor to it: we can store any kind of data, since we specify no kind of data.

Note that some containers require some kind of data comparability; for example, dictionaries require at least the ability to compare two keys to see if they are the same, and binary search trees require the ability to compare two keys for equality and less or greater than. Zen data structures can be adapted to work with these (*cf* the **qsort()** function) at the expense of complicating the code.

From the user's perspective, zen data structures differ from other structures in two main ways:

- When initializing the container, the number of bytes to move must be given.
- Adding to and removing from the container require pointers to data be passed.

To try this out, you will work with the following files:

- `zenStack.h`
- `zenStack.cc`
- `stackTest2.cc` — same as stackTest1.cc

In `stackTest2.cc`, change the stack declaration to

```
ZenStack myStack(sizeof(int));
```

This determines the number of bytes in an integer, and then tells the zen stack to move that many bytes with each **push()** and **pop()**.

Next, change the **push()** call to

```
myStack.push(&d);
```

Finally, change the **pop()** line to

```
myStack.pop(&d);
```

Compile `zenStack.cc` and `stackTest2.cc` with one `g++` command. The resulting executable should work exactly as the first one did.

Method 3 — Multiple classes

A third method is to create a separate class for each type of data the container needs to store. Although we could actually write out multiple classes, there is a simpler way to accomplish this — the **template**.

With a template, a placeholder is used in the class definition wherever the data type is needed. When a template container object is declared, the actual data type is specified. The compiler then makes a behind-the-scenes copy of the template class, with the placeholder replaced with the actual data type.

One drawback of this approach is that all code must be placed inside the class. Non-template containers are usually split into an interface file holding the class definition and animplementation file holding the code for the class methods. Since the compiler needs all of the class information in order to make copies, the code must be part of the template.

To try this method out, you will work with the following files:

- `stack.h`, which is `simpleStack.h` and `simpleStack.cc` combined into one file
- `stackTest3.cc`, which is the same as the other two test files

In `stack.h`, remove the **typedef** line, and add the following line immediately above the class line:

```
template <class StackType>
```

This establishes the class as a template, using **StackType** as a placeholder.

Next, in `stackTest3.cc`, change the stack declaration to

```
Stack<int> myStack;
```

That's all. . . compile `stackTest3.cc` and it should work just as the other two files.

Method 4 — Shallow copy

There is a fourth method to create generic containers, but we will not attempt it in this lab.

The idea is for the container to only store pointers to data elements, not the elements themselves. This is called *shallow copying*. The pointers or references can be made generic enough to store any kind of data in the container; one container can even store multiple types of data simultaneously — a limitation on template containers. However, care must be taken to ensure that each item stored in this manner is actually a unique element and not just multiple pointers to the same object.

What to turn in

Please turn in your modified source code.