T e m p l a t e s

Savitch chapter 16

Templates: Introduction

- Templates are used if a stand-alone function will work in exactly the same way no matter what is the type of its arguments.
  - e.g. swap 2 integers, swap 2 floats, swap 2 characters …
- Or if a class will operate in the same way no matter what is the type of its data members
  - e.g a linked list of integers, a linked list of doubles, a linked list of strings ….
- Templates allow you to create reusable code.

Function templates

the following function computes the sum of the elements of an array of integers

```
int sumArray(int *ary, int sz)
{
   int sum = 0;
   for (int idx = 0; idx < sz;
   idx++)
    sum += ary[idx];
   return sum;
}
```

The functions are identical, except for changing the type of the array and the return type from `int` to `double`.

Here is the same function 'overloaded' for use with an array of doubles

```
double sumArray(double *ary, int sz)
{
   double sum = 0;
   for (int idx = 0; idx < sz; idx++)
    sum += ary[idx];
   return sum;
}
```

Many algorithms like this are logically the same no matter what type of data is being operated upon.

We could write the function in a generic way if we could say "This is how you do it for any type T"

Generic Data Types

```
template <typename T> T sumArray(T *ary, int  sz)
{
  T sum = 0;
  for (int idx = 0; idx < sz; idx++)
   sum += ary[idx];
  return sum;
}
```

This is how you do it for any type T

- To make the function completely generic, so that we can use it with any type we precede it with a template declaration.
  - We then have a template for the function using generic data types without having to explicitly recode a specific version for each different data type.
  - The appropriate data type will be plugged in when needed
  - But in this example, the operator += used in the function must have a meaning for every type we use. (*Beware - here be dragons*!)
- Thus, templates allow you to create re-usable code - *code once, the compiler will re-use for many different types*.

Template function syntax

- A template function is created using the keyword **template** .

```
template <class T> T sumArray(T *ary, int sz)
{
   T sum = 0;
   for (int idx = 0; idx < size; idx++)
    sum += array[idx];
   return sum;
}
```

- The keyword `typename` may be replaced with the keyword `class`, as above: both have the same meaning in this context. `typename` is probably clearer, though you will see both, and `class` is possibly more common.
- The **template** portion of a template function does not have to be on the same line as the function's name. It is more usually on a line of its own.
  - But the function name is a continuation of the template statement, so no semi-colon.

```
template <typename T>;
T sumArray(T *ary, int sz)
{…..etc;
```

An error

Template function: how does it work

- A template is really a recipe from which real instances of the function can be constructed by the compiler as they are needed.
- When the compiler constructs a specific version of a template function, it is said to have created a *generated function*. The process of generating a function is referred to as *instantiating* it. So a generated function is a specific instance of a template function.

```
int main()
{
   int array1[] = {2, 4, 6, 8};
   double array2[] = {1.1, 2.2, 3.3, 4.5};
   cout << sumArray(array1, 4) << '\t';
   cout << sumArray(array2, 4) << endl;
   return 0;
}
```

output:

```
20  11.1
```

When it sees this code, the compiler will generate two instances of the `sumArray` function - one to use with `ints`, and one to use with `doubles`

Multiple Generic Types

You can define more than one generic data type with the **template** statement, using a comma-separated list. For example, the next program creates a generic function that has two generic types:

```
#include <iostream>
using namespace std;
template <class T1, class T2>
void spaceOutput(T1 x, T2 y)
{
   cout << x << ' ' << y << endl;
}
```

```
int main()
{
   spaceOutput(10, "hi");
   spaceOutput (0.23, 10L);
   return 0;
}
```

In this example, the placeholder types **T1** and **T2** are replaced by the compiler with the data types `int` and `char*` and `double` and `long`, respectively, when the compiler generates the specific instances of `spaceOutput ()`

Writing Template Functions

- When you write a template function you're probably better off starting with a normal function that works on a fixed type.

- You can design and debug it without having to worry about template syntax.

- Then, when everything works properly, you can turn the function definition into a template and check that it works for additional types.

- The template code for the function and the template definition can go into a header file if you wish
  - Note that the template code you write is not itself compiled – actual code is generated from it by the compiler for any types needed.

Examples to try

- Write a template function `swapArgs` to swap the values in 2 variables of the same type
  - Test with the following code

```
int x = 27, y = 45;
swapArgs(x, y)
cout << x << '\t' << y << endl;

string line1 = "How are you doing?", line2 = "I'm
    doing fine!";
cout << line1 << ' ' << line2 << endl;
swapArgs(line1, line2);
cout << line1 << ' ' << line2 << endl;
```

- Write a template function to search an array of *any* type for a particular value, and return the index position of the value
  - Provide your own test code

Template classes

- Recall the stack data structure. The stack may be a stack of integers, or a stack of floats, or a stack of characters, or ….. We implement the stack as a class, thus encapsulating its methods, and with the use of templates, we can construct a template class for a stack which can be used to hold any (single) type of element.

- The stack class will have methods to push an element onto the stack, and to pop from the stack the last element pushed. These methods might have signatures as follows:

```
bool push(element-type);
bool pop (element-type&);
bool empty();
```

- We will look at an implementation of a stack as a class template which will work for a stack of *any type of elements*.
- We will use an array to implement the class
  - of course we could also implement with a linked list, but the array makes a simpler first example of a class template as it only involves one class (The linked list would need a node class as well as the list class)

Stack class template

- Must keep tabs on the index position of the last item added (the 'top' of the stack)
- and the size of the array (max no of elements which can be held by the stack)
- We will use a private method to indicate if the stack is full (i.e. size of array has been reached) **`bool full() ;`**
  - if this was a linked list implementation, the code for full() would be trivial - why?

```
template<typename ElementType>
class Stack {
public:
   Stack (int = 10); //default stack size is 10
   ~Stack();
   bool push(ElementType el);
   bool pop(ElementType & el);
   bool empty();
private:
   ElementType *stkptr;    //to hold the array of elements
   int top;   //-1 if the stack is empty
   int capacity;     // the no of elements which the stack can hold

   bool full();
};
```

Definition of the class template

```
template<typename ElementType>
class Stack {
public:
   Stack (int = 10); //default stack size is 10 in our class
   ~Stack();
   bool push(ElementType );

   bool pop(ElementType &);
   bool empty();


private:
   ElementType *stkptr;   //to hold the array of elements
                         // making up the stack
   int top;              //-1 if the stack is empty
   int capacity;     // the no of elements which the stack can hold
    bool full();
};
```

Specifying template classes to be instantiated by the compiler

- Once the class template has been defined, we can specify that we wish the compiler to use it to generate various template classes, by declaring instances of these classes

  - a stack of integers to hold up to 5 integers

```
Stack<int> intStack(5);
```

  - A stack of doubles, to hold up to 10 doubles

```
Stack<double> dblStack(10);
```

  - And then use these instances

```
intStack.push(12);
dblStack.push(4.5);
```

Defining the class methods

For each method of the class template, we must re-iterate the template keyword

```
template <typename ElTyp>
Stack<ElTyp>::Stack (int s)
{
   if (s < 0)
        capacity = 10;
   else
        capacity = s;
   top = -1;  // starts empty
   stkptr = new ElTyp[capacity];
}

template <typename ElTyp>
Stack<ElTyp>::~Stack()
{
   delete [] stkptr;
}

template <typename ElTyp>
bool Stack<ElTyp>::full() {
   return (top == capacity - 1);
}
```

```
template <typename ElTyp>
bool Stack<ElTyp>::empty() {
   return (top == -1);
}

template <typename ElTyp>
bool Stack<ElTyp>::push(const ElTyp el)
{
   if (!full() ) {
    stkptr[++top] = el;
    return true;
   }
   return false;
}

template <typename ElTyp>
bool Stack<ElTyp>::pop(ElTyp &poppedEl)
{
    if (!empty()) {
    poppedEl = stkptr[top--];
    return true;
   }
   return false;
}
```

Remember, we could use the keyword `class` instead of `typename`

Non-type parameters in class templates

- It is possible to use other parameters in the template header:
  `template <typename elementType, int size> class Stack;`
- *Here, an instance of a stack of doubles of size 100 could be declared in the code as* `Stack<double, 100> mystack;`
  - *At compile time*, a class consisting of a stack of doubles of size 100 would be generated and compiled.

```
template<typename ElType, int capacity> class Stack {
public:
   Stack (); //no default - the capacity is decided
   ~Stack();
   bool push(const ElType el);
   bool pop(ElType &);
   bool empty();

private:
   ElType stkptr[capacity];
   int top; //-1 if the stack is empty
   // int capacity;  // not needed now - size is set already
   bool full();
};
```

the size of the stack is determined at compile-time, eliminating the need to allocate dynamically

Using friend classes in class templates

- A friend class declared in a class template might be
  - a standard class,
  - a particular template class (i.e. the type is specified)

```
template<typename X>
class ListNode {
public:
    friend class List<X>;
private:
    X thevalue;
    ListNode<X> *next;
};
```

If we used a different name for the type, then Lists of any type would be friends of the `ListNode`.
Here it is just the `List` *of the same type* as the `ListNode`

```
template<typename X>
class List {
public:
    //the methods go here
private:
    ListNode<X> *head;
}
```