CSC2059 Data Structures and Algorithms

Practical 4: Template Functions and Classes

Wednesday 20th October 2021

Launch Visual Studio. Create a new Solution Practical4 and name the first project Template1.

Project 1. Template functions

The following shows a C++ implementation of a function (see Project6 in Practical 2), for inserting a value val into an integer array pointed to by pi of size elements, at position pos:

```
bool bInsert(int*& pi, int& size, int pos, int val)
{
	if (pos < 0 || pos > size) {
		cout << "pos is out of range" << endl;
		return false;
	}

	// new array size after insertion
	size++;
	// new array
	int* piNew = new int[size];
	if (piNew == NULL)
		return false;

	// copt pi to piNew & insert val
	for (int i = 0; i < pos; i++)
		piNew[i] = pi[i];
	piNew[pos] = val;
	for (int i = pos + 1; i < size; i++)
		piNew[i] = pi[i - 1];

	// delete old array
	delete[] pi;
	// point pi to the new array
	pi = piNew;

	return true;
}
```

You are asked to convert the above function into a template function so that it can be used to insert an element for other arrays of suitable data types. To do this you may:

(1) In the project Template1 create a source file test1.cpp, in which write your template function bInsert.

(2) Below the function, write a main function to perform suitable tests. For example,

1. Define size= 5. Allocate a string array pointed to by pstrs with size elements. Assume initialising the array with the following string elements. Print the values in this array after the initialisation.

A `string` array with size = 5 elements

abc
def
jkl
mno
pqr

Then, call the above function with the array to insert a string: “ghi”, at position 2. Note that when you use “ghi” directly in place of val Visual Studio may give you an error message of type mismatch, as VS takes “ghi” as a constant and val as a variable. You can convert “ghi” to a variable by using an explicit type conversion (string)”ghi”, or by using string(“ghi”) instead of “ghi”. After the insertion operation print the values in the new array to verify that the function is correct.

2. Class is a data type defined by the user and can certainly be represented by the generic data type T. Download the provided Person class (Person.h, Person.cpp) and add this class into the Template1 project. Define size = 10 and allocate a Person array pointed to by pps with size person records. Print the values in this array using the member function print of this class.

   Then, call the above bInsert function with the array to insert a new Person object with first/family/ID/birth = “Jack”/“Smith”/“123456”/1990 at a suitable position (e.g. 4). After the insertion operation print the values in the new array to verify that the result is correct.

Project 2. Template class

In C++, it is possible to overrun (or underrun) an array boundary at run time without generating a runtime error message. The following shows a class that implements an int type *safe* array that provides runtime boundary checking by overloading the [ ] operator. Study the class and the attached test program carefully:

- The class uses a constructor to allocate the memory data for the array and a destructor to free the memory after the array object is destroyed;
- The class overloads the operator [ ] for accessing the individual elements of the array with boundary checking;
- The operator function returns the address (i.e. the reference) of the element that is being accessed, so that the [ ] operator can be used on both the left side and the right side of an assignment statement (see the attached main program for examples).

Safe array class & testing program

```
class sarray {
public:
	sarray(int size);
	~sarray();

	int& operator[](int i);

private:
	int size;
	int* data;
};
```

```
sarray::sarray(int size)
	: size(size)
{
	if (size > 0) data = new int[size];
	else {
		cout << "illegal array size = " << size << endl;
		exit(1);
	}
}

sarray::~sarray()
{
	delete[] data;
}

int& sarray::operator[](int i)
{
	if (i < 0 || i >= size) {
		cout << "index " << i << " is out of bounds." << endl;
		exit(1);
	}
	return data[i];
}

int main()
{
	// create a 10-element safe array
	sarray array(10);

	// in-bound access, [] is used on the left side or an assignment
	array[5] = 23;
	// in-bound access, [] is used on the right side of an operation
	cout << array[5] << endl;

	// out-of-bound accesses
	array[13] = 392;
	cout << array[-1] << endl;

	return 0;
}
```

In the above program, comment out the lines highlighted in yellow, and run the program in release mode (Solution Configurations->Release). You may see that C++ can 'tolerate' some out-of-bounds errors. Do not forget to uncomment the lines after this test.

The above class, however, only allows you to create safe arrays for integers. In this project, you are asked to implement a template class, to create a generic safe-array class that can be used for creating safe arrays for any suitable data types.

In the Solution Practical4 add a new project Template2, in which you implement the template class for safe arrays in a header file sarray.h and conduct suitable tests of the class you implement in a source file test2.cpp.

To test your new class you should create safe arrays for at least three data types: double, char, and string. You should test your code by accessing the arrays you create within bounds and out-of-bounds.

Project 3. Template Stack class

In the Solution Practical4 add a new project Template3. Download the provided template classes StackNode.h and Stack.h and add them into the new project.

(1) First, add a new operator == function inside the Stack class to perform logic comparison between two Stack objects e.g. a == b. By definition, two stacks are identical if and only if they have the same size and same sequence of data items (assuming that different data items are comparable for equality). The comparison should not change either of the stacks in any way.

Create a test3.cpp in the project, with a main function to test the operator function by creating two identical stacks a, b and two different stacks a, b, respectively, for comparison:

```
if (a == b)
        cout << "Two stacks are identical" << endl;
else
        cout << "Two stacks are different" << endl;
```

(2) Next, extend Project 3 of Practical 3 from the iStack class to the template Stack class, to add member functions print and search into the class. The tasks were detailed in (1) and (2) in Project 3, Practical 3, for the iStack class; now you will implement the same functions for the template Stack class.

(3) Finally, extend the tasks described in (3) and (4) in Project 3, Practical 3 from testing the iStack class to testing the template Stack class. You will test the new member functions for the template Stack class by using a suitable data type of your choice.

If you have any questions, ask a demonstrator or a teaching staff face to face in the labs, or online by using MS Teams, during the timetabled practical session time. At any other time outside the timetabled session time, you can send in your questions by using the Ticketing System, or by using emails.