CSC2040 Data Structures, Algorithms and Programming Languages

Practical 2

Friday 14th October 2016

This practical will introduce you to arrays and pointers. One of the main differences of arrays in C++ and java is that there is no bounds checking in C++. Your code can wrongly index an array of size N beyond N without necessarily generating compile time or run time errors, leaving a deadly bug that may strike at an unexpected time. Therefore it is up to you the programmer to ensure that you prevent array overruns in the first place.

Program 1 Create and use an int array, and pointer-array equivalence

Create a new C++ file called Practical2.cpp. As in last week's practical class create a main function. Type in the following code:

```
int sample[10]; // this reserves 10 integer elements

//load the array
for (int t = 0; t < 10; t++)    //for loops are useful to process arrays
      sample[t] = t;

//display the array
for (int t = 0; t < 10; ++t)
      cout << "This is sample [" << t << "]: " << sample[t] << endl;
return 0;
```

Save and run the program. Note how the array is created and used. This should be familiar to you.

An array can be initialised in the definition. Replace the above array definition with the following and run the code:

```
int sample[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

//display the array
for (int t = 0; t < 10; ++t)
      cout << "This is sample [" << t << "]: " << sample[t] << endl;
```

In C++, pointers and arrays are tightly related. An array's name is a pointer to the 1st element of the array. So an array can be accessed using pointer arithmetic, and a pointer can be accessed using array indexing. In the above program, before `return 0;` add the following pointer-based access of the array and run the code:

```
// define a pointer to the start of the array using the array's name
int* p = sample;

//load the array using pointer arithmetic
for (int t = 0; t < 10; t++)
      *p++ = t * t;

//display the array
p = sample;
for (int t = 0; t < 10; ++t)
      cout << "This is sample [" << t << "]: " << *p++ << endl;
```

Note that after loading the array using the pointer arithmetic, you must reset the pointer to the start of `sample` before it can be used to print the array correctly.

Program 2 More pointers

Pointers are one of the most powerful features of C++. Try to understand how to use the pointer operators * and &, and the NULL pointer, and how to program with pointers. Type in the following code into a main function.

```
int i = 27;
int *i_ptr = &i; // declare a pointer and give it the value of the address of i
cout << "value in i is " << i << endl;
cout << "address of i is " << &i <<endl;
cout << "value in pointer is " << i_ptr << endl;
cout << "value from dereferencing pointer is " << *i_ptr << endl;

*i_ptr = 35;     // change the value in i through pointer
cout << "value in i is " << i << endl;

double* d_ptr = NULL; // declare a pointer and initialise it to a NULL pointer
cout << "value in pointer is " << d_ptr << endl;

*d_ptr = 0.25;   // error! NULL is not a valid memory address
```

Run the program and ensure you understand the meaning of each of the operators and program statements by answering the following:

What do you notice about the value of i and *i_ptr? Can you explain this?

What do you notice about the value of &i and i_ptr? Can you explain this?

The following program tries to print the value of `x`, which is 10, but is wrong. Type in the code into a main function and fix the bug.

```
int x = 10;
int* p = NULL;

*p = x;
cout << *p << endl;
```

The next program illustrates an example of dangling pointers that do not point to a valid object of the appropriate type (these are special cases of memory safety violations). Type in the code into a main function and trace the code line by line in the Debug mode, to gain a first-hand experience of the problem.

```
int* p = NULL;

if(p == NULL) {
   int x = 10;
   p = &x;
}
// x falls out of scope ("x" is undefined)
// p is now a dangling pointer
*p = 3;
```

Program 3 Functions passing array arguments by pointers

Because of the pointer-array equivalence, it is common to pass array arguments by using pointers.

You are asked to write code for a function

```
bool same_elements(int* a, int* b, int n)
```

that checks whether two arrays (represented by two pointers) of the same size `n` have the same elements with the same multiplicities. For example, two arrays

```
121     144     19      161     19      144     19      11
```

and

```
11      121     144     19      161     19      144     19
```

would be considered to have the same elements because 19 appears 3 times in each array, 144 appears twice in each array, and all the other elements appear once in each array. The function returns a bool value – `true` or `false`, identical to `1` or `0`, depending on the test result.

Points to note about functions:

All C++ functions share a common form, which is shown here:

```
return-type name(parameter-list)
{
//body of function
}
```

Declare the function in an appropriate header file and define the code of the function in a corresponding source file. Then create a main function to run the test, in which you define two arrays and use them to call the function, to test if they have the same or different elements based on the return value.

Program 4 Dynamic memory allocation

Dynamic allocation is an important part of almost all real-world programs. C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time. This is your responsibility in C++, unlike in java where a garbage collector automatically removes all objects that are no longer needed.

In this exercise, we will practice the use of dynamic allocation. We will use the header-source file structure.

Below are two function definitions to be placed in a header(.h) file:

```
void insert(int* &array, int &array_len, int pos, int val);
void remove(int* &array, int &array_len, int pos)
```

The function`insert` facilitates the insertion of a value `val` at position `pos` in the given array of size `array_len.`

The function `remove` facititates the removal of the value at position `pos` in the given array of size `array_len.`

Implement the functions in a source file(.cpp), using dynamic allocation inside the two functions to allocate temporary working memory and memory for the new (value inserted or removed) arrays. Below are a list of instructions to help you achieve this:

- Check that the array exists
- Check the position of insertion is a valid position i.e. within the range of the array
- Increment `array_len`
- Create a new array *// temporary working memory*
- Transfer everything up to `pos` to the new array
- Insert `val` into the new array at position `pos`
- Transfer the remainder of the original array to the new array
- Delete the original array and set the new array as the original array

Based on the ideas above implement the remove method yourself. Call the functions in a main function, by supplying the appropriate arrays and parameter values.

Make sure you understand why the Call-by-reference method is used to pass the array `array` and array length `array_len` arguments.

Show your work to a demonstrator before leaving class today.