Courses/CS 2124/Lab Manual/Arrays and C-Style Strings

Introduction

The purpose of this lab is to introduce the fundamental data structure known as an *array*, and compare it with the more advanced object-oriented implementation of `std::vector`.

Array of integers

The *array* data structure is defined as a collection of objects stored sequentially in memory (one after another). The array must support efficient *random-access* of its elements (meaning that you can directly access any element in the array efficiently). In C++, it is required that all objects in the array must be the same type, and C++ guarantees that the objects are stored *contiguously* (sequentially with no other data between elements).

Array declarations look like this:

```
int myArray[10];
```

Where `int` refers to the type of values contained in the array (any type that can be *default-constructed* is allowed). `myArray` is just an identifier as the name of the array. The `10` in the square brackets is the *size declarator*, and must be either a literal or constant integer. The size declarator tells the compiler how many elements your array will contain. The square brackets themselves are the *array type modifier* in this context; they signal that the variable you are declaring will be an array.

Let's perform some simple exercises using arrays of various types.

Create a main program `main.cpp` and include the `<iostream>` and `<cstdlib>` libraries. Just after the function header for main, seed the `cstdlib` pseudo-random number generator with the following line:

```
srand(42);
```

This will allow you to generate a sequence of numbers that is reproducible so that your program will generate the same pseudo-random sequence every time you run the program. For more information on seeding the PRNG see: http://www.cplusplus.com/reference/cstdlib/srand/

1. Create an array of 100 integers.
2. Create a function `fill_random()` that will take the array and its size as parameters and assign a random number between 1 and 1000 to each element using the following expression:
   - `rand() % 1000 + 1`
3. Create a function `average()` to calculate and return the average (as a floating-point value) of the array. Add code to your main program to call the function with the array you created in step 1, and print the result to the screen.
4. Create a 2-dimensional array with 15 rows and 15 columns, capable of storing characters.
5. Write a function `make_puzzle()` that will take the 2-D array and the number of rows it contains as parameters. The function should fill the array with random characters by assigning values between `'a'` and `'z'` using the following expression:
   - `static_cast<char>(rand() % 26 + 'a')`
6. Write a function `print_puzzle()` that will take the 2-D array and the number of rows it contains as parameters. The function should print the array to the screen, separating the characters with one space and placing a newline after each row. (The output should appear square.)
7. Move the function prototypes and definitions you created into a new header file named `array_functions.h`. Be sure to protect the contents of the header file with include guards, and include it in your main program. Test that your code still works.

Array of Rectangles

Use the following definition of a class `Rectangle`; place it in a file called `Rectangle.h`. You will need to include it in both your main program and in `array_functions.h`.

```
class Rectangle{
public:
    Rectangle() = default;                              // Use implicit defaults from initializers
    Rectangle(int l, int w) : length{l}, width{w} {  }  // or instantiate with dimensions.
    void set_length(int l)  { length = l;            }  // Thin mutators for the length
    void set_width (int w)  { width  = w;            }  // and width.  Better ones would validate!
```

```
    int  get_length() const { return length;         } // Inline accessors for length,
    int  get_width () const { return width;          } // width
    int  get_area  () const { return length * width; } // and area.

private:
    int length = 1;                                   // length and width default to 1
    int width  = 1;                                   // to create a "unit square"
};
```

The 2-parameter constructor for `Rectangle` makes use of a *constructor member initializer list*. If you have not seen this syntax before, you can find more information at: http://en.cppreference.com/w/cpp/language/initializer_list

1. Create an array of 20 `Rectangle` objects. Because array declarations cause default construction, all of the rectangles will be unit rectangles (1×1) initially.
2. Create a function in `array_functions.h` called `print_rectangles()` that will take the array of rectangles and its size as parameters and print the dimensions of each rectangle to the screen, one per line, in the format `1: 1 x 1`, where the first number is the element number corresponding to rectangle being printed (counting from one), the second number is the `length` of the rectangle, the 'x' is literally the character `'x'`, and the third number is the `width` of the rectangle being printed. Verify that your function works before moving on, but comment out the call you have tested.
3. Create a function in `array_functions.h` called `random_rectangles()` that will take the array of rectangles and its size as parameters and set the length and width of each rectangle object to a random value computed by the expression:
    - `rand() % 100 + 1`
4. Print the rectangles to the screen using the `print_rectangles()` function.

C-Style Strings

Native strings in C++ (often called "C-style strings") are just sentinel-delimited arrays of characters, where the sentinel is the "null character" `'¥0'` and must appear immediately following the last character in the logical string. For example, the string `"Hello"` corresponds to the array:

```
char example[]{ 'H', 'e', 'l', 'l', 'o', '¥0' };
```

But C++ has (some) special built-in support for C-style strings, so we can simplify the initialization expression shown above to:

```
char example[]{ "Hello" };
```

Add the following lines to your program, then include the `<cstring>` library and compile and run to see the output.

```
char state[]{ "Arkansas" };
std::cout << state << " has " << strlen(state)
          << " (logical) characters and occupies "
          << sizeof(state) << " bytes.¥n";
```

Was the answer what you would expect? (Keep in mind the size of a `char` in C++ is 1 byte.) You should see that the *physical* size of the string is 1 byte larger than its *logical* size.

1. Create a new header file `c_string_utils.h`. Add include guards and include it in your main program.
2. Create a function `my_strlen()` in `c_string_utils.h` that will take a C-style string as its only argument and will return the logical length of the string. You must compute the length using a *sentinel-controlled loop* of your own; you may not use the `<cstring>` library for this.
3. Repeat the code above in your main program, except that you should now call `my_strlen()` instead of `strlen()`. Verify that the two functions give the same results.

Be sure to test your code as you go — compile early and often!

When you complete the assignment, zip all source code files and submit the archive as **OOP_lab02_inlab.zip**.