

Pointer Exercises 1: Pointers and Arrays

Each exercise should be done in a separate function. For example, Exercise 1 should be programmed in the following format:

```
void Exercisel() {
    cout << "Exercise 1" << endl << endl;

    // Enter the Exercise 1 code here

    PressReturn("Exercise 1 Complete");
}
```

In the `main` function, call this function once via:

```
Exercisel();
```

In addition to these *Exercise* functions, we will ask you to define other functions. These should be placed before the functions that require them. One of the functions that you should define is the following `Hex` function that will allow you to print pointers:

```
string Hex(void* p) {
    stringstream stream;

    stream << hex << setfill('0') << setw(8) << (long) p << ends;

    return stream.str();
}
```

`Hex` takes a pointer variable `p` and returns the 8 character hexadecimal string that represents its contents which is an address value. To print the contents of `p`, use:

```
cout << Hex(p);
```

Exercise 1: Introduce `int` variables `x` and `y` and `int*` pointer variables `p` and `q`. Set `x` to 2, `y` to 8, `p` to the address of `x`, and `q` to the address of `y`. Then print the following information:

- (1) The address of `x` and the value of `x`.
- (2) The value of `p` and the value of `*p`.
- (3) The address of `y` and the value of `y`.
- (4) The value of `q` and the value of `*q`.
- (5) The address of `p` (not its contents!).
- (6) The address of `q` (not its contents!).

Use the `Hex` function to print all pointer/address values and format the output so it is easy to make comparisons.

Exercise 2: Introduce `int` variables `x`, `y`, `z` and `int*` pointer variables `p`, `q`, `r`. Set `x`, `y`, `z` to three distinct values. Set `p`, `q`, `r` to the addresses of `x`, `y`, `z` respectively.

- (1) Print with labels the values of `x`, `y`, `z`, `p`, `q`, `r`, `*p`, `*q`, `*r`.
- (2) Print the message: Swapping values.
- (3) Execute the swap code: `z = x; x = y; y = z;`
- (4) Print with labels the values of `x`, `y`, `z`, `p`, `q`, `r`, `*p`, `*q`, `*r`.

Exercise 3: Introduce `int` variables `x`, `y`, `z` and `int*` pointer variables `p`, `q`, `r`. Set `x`, `y`, `z` to three distinct values. Set `p`, `q`, `r` to the addresses of `x`, `y`, `z` respectively.

- (1) Print with labels the values of `x`, `y`, `z`, `p`, `q`, `r`, `*p`, `*q`, `*r`.
- (2) Print the message: Swapping pointers.
- (3) Execute the swap code: `r = p; p = q; q = r;`
- (4) Print with labels the values of `x`, `y`, `z`, `p`, `q`, `r`, `*p`, `*q`, `*r`.

Paper-and-pencil-exercise: Draw diagrams to explain the results of Exercises 2 and 3.

Exercise 4: Introduce 4 `int` variables and one array variable `a` as follows:

```
int x = 11;
int y = 12;
int a[5];
int u = 31;
int v = 32;
```

Then:

- (1) Write a loop to fill the array `a` with 21, 22, 23, 24, 25.
- (2) Execute the following loop:

```
for (int i = -2; i < 7; i++)
    cout << setw(2) << i << " " << setw(2) << a[i] << endl;
```

Note: *The above loop represents very bad practice* since it accesses two cells before the start of the array and two cells after it. The loop shows that you will access adjacent memory and that C++ gives no direct protection against such memory access errors.

Exercise 5: Define an array `int a[5]` and fill this array with values 1, 4, 7, 10, 13. Introduce an `int i` and an `int*` pointer variable `p`. Then run two printing loops:

```
for (i = 0; i < 5; i++)
    cout << i << " " << Hex(a+i) << " " << a[i] << endl;

cout << endl;
i = 0;
p = a;

while (p < (a+5)) {
    cout << i << " " << Hex(p) << " " << *p << endl;
    i++;
    p++;
}
```

Exercise 6: Define an `int*` pointer variable `a`. Then:

- (1) Use `new` to make `a` point to a dynamic array of 5 cells of type `int`.
- (2) Write a loop to fill `a` with values 3, 7, 11, 15, 19.
- (3) Using `Hex`, print the pointer address stored in `a`.
- (4) Write a loop to print the values in `a` with one cell per line.
- (5) Delete the dynamic memory allocated to `a` using `delete []`.

In the next exercises, you will be asked to enter the size of a dynamic array at runtime. You should structure each exercise using the following example as a model:

```
void Exercise7() {
    cout << "Exercise 7" << endl << endl;

    int size;

    while (ReadingInt("Array Size", size)) {
        if (size < 1) {
            cout << "Array Size Must Be At Least 1" << endl << endl;
            continue; // continue jumps back to loop condition
        }

        // Enter the main Exercise 7 code here
    }

    PressReturn("Exercise 7 Complete");
}
```

The `while` loop will permit the tests in each exercise to be executed several times. If you decline to provide a `size` value at the `Array size` prompt, the loop will terminate.

These exercises will also use *random numbers*, that is, numbers generated by an algorithm that attempts to make the numbers generated appear to be random. To obtain random numbers, use an object of the `RandomNumber` class. Here's how such an object may be used:

```
RandomNumber R; // define a RandomNumber object R
x = R.RandomLong(1, 99); // store in x a random number between 1 and 99
```

Exercise 7: The main work in this exercise (inside the `while` loop) is as follows:

- (1) Define an `int*` pointer variable `a`.
- (2) Use `new` to make `a` point to a dynamic array of `size` cells of type `int`.
- (3) Define a random number object `R`.
- (4) Fill the `size` cells of `a` with random numbers between 1 and 99.
- (5) Using `Hex`, print the pointer address stored in `a`.
- (6) Write a loop to print the values in `a` with one cell per line.
- (7) Delete the dynamic memory allocated to `a` using `delete []`.

Note: As a practical matter, the console window has only 25 lines so choose `size <= 20` during testing.

Exercise 8: In this exercise, you will repeat the work of Exercise 7 and will add graphics output. You may certainly copy the code from Exercise 7 as a starting point. The graphics will be a bar chart created from the data in the array `a`. You will need to build a separate function:

```
void SimpleBarChart(int* a, int size)
```

This function will be simple in the sense that it makes simplifying assumptions, namely:

- (1) $1 \leq \text{size} \leq 29$
- (2) For all i : $1 \leq a[i] \leq 99$

The function `SimpleBarChart` will draw one bar (in the 300 x 300 graphics window) for each array cell `a[i]`. To do this, *you will have a loop* on the index i that will call:

```
FillRect(x1, y1, x2, y2);
```

Before calling this graphics function, you must of course specify the 4 `int` variables. The lower left corner ($x1, y1$) is specified as follows:

The value of $x1$ is determined by the convention on the bars. Each bar will be 5 pixels thick and will have a space of 5 pixels before the next bar. Assuming that the first bar starts at the position $x1 = 5$ then in general for the i -th bar: $x1 = 5 + 10*i$.

The value of $y1$ is the lower limit of the screen window: $y1 = 300$.

The upper right corner ($x2, y2$) is then computed as follows:

The value of $x2$ is 5 more than $x1$.

The value of $y2$ is $y1$ minus 3 times the value of `a[i]`.

Two more issues: To make sure that the graphics window is erased, you must call

```
ClearDrawing();
```

at the beginning of `SimpleBarChart`. If you want the bars to be filled with a random color rather than with black then add the following lines immediately after the `ClearDrawing()` call.

```
RandomNumber S;  
SetFillColor(S.RandomLong(0,255), S.RandomLong(0,255), S.RandomLong(0,255));
```

Once you have completed the separate function `SimpleBarChart`, you should call this function in Exercise 8 as a replacement for Steps 5 and 6 of Exercise 7. If you wish, you can ask the user (via a `confirm` call) whether to print the array data (but this should not be the default choice).

Exercise 9: In this exercise, you will use the techniques of the previous problems to create and draw a random polygon. Since a polygon requires at least 3 corners (vertices), you should ensure that the `size` parameter supplied by the user is 3 or larger.

You will use the `PointData` class to collect the random polygon vertices. All you need to know about the `PointData` class is:

(1) It is possible to build dynamic arrays of `PointData` via:

```
PointData a = new PointData[size];
```

(2) To set the coordinate values `x`, `y` of a `PointData` object `P` use:

```
P.Set(x, y);
```

The implication of this is that the `i`-th element in the `PointData` array `a` can be set via:

```
a[i].Set(x, y);
```

Since we wish you to draw a random polygon, you will need to use a `RandomNumber` object to set the `x`, `y` values. The range should be `0, 299` which is the graphics window size.

So, after creating the `PointData` array `a`, you need to use a loop to fill in the `PointData` cells with random `x`, `y` values.

To draw the polygon, create a function:

```
void SimplePoly(PointData* a, int size)
```

This function should:

- (1) Call `clearDrawing()`.
- (2) Use the `MoveTo` function to move to the `0`-th point in array `a`.
- (3) Use a loop and the `LineTo` function to draw a line to each succeeding point in array `a`.
- (4) Close the polygon with `LineTo(a[0])`.

The overall structure of Exercise 9 is similar to Exercise 8 and we leave the details to you. If you want random colors for the polygon edges, use `setPencolor` in a manner similar to what is done in Exercise 8.

Exercise 10: In this exercise, you will modify Exercise 8 by sorting the random array that you create and by providing graphics feedback for the sort.

Introduce two `int*` pointer variable `a` and `b`. Create two dynamic arrays of the specified `size` and store the pointers in `a` and `b`. As in Exercise 8, fill `a` with random values from 1 to 99. Then copy the contents of `a` to `b`. You will do the sorting on the copy `b` rather than on the original `a`.

To do the sort, define a function

```
void MySort(int* a, int size)
```

Do not be confused. Although the formal parameter of `MySort` is `a`, you will in fact pass your variable `b` to this function when it is called.

In the function `MySort`, you should program any algorithm you wish to sort the array values. It is not important at this stage that the algorithm be extremely efficient because later on in the course you will learn about efficient sorting algorithms. What is important now is that you program a *correct* sorting algorithm and that you begin to think about sorting as an issue.

In Exercise 10, after you copy the `a` values to `b`, you should call:

```
MySort(b, size);
```

Then it is time for graphics feedback. Here you will use the two graphics windows to show the original array `a` and the sorted array `b`. To do this, make the calls:

```
SetGraphicsWindow(0);
SimpleBarChart(a, size);

SetGraphicsWindow(1);
SimpleBarChart(b, size);
```

Finally, if you wish, you can provide an option to the user to print the array data. We suggest three columns with the index `i`, the value `a[i]`, and the value `b[i]`.

At the end of the code for Exercise 10, don't forget that you must now delete two arrays:

```
delete [ ] a;
delete [ ] b;
```