

AST21105 Object-Oriented Programming & Design

Lab 4 – Pointers and Dynamic Allocation

A. Submission Details

In this lab, you are required to submit **ONE** C++ program to solve the given problem shown in the section “Find Prime Numbers”. To make the program implementation easier, you are suggested to write your program using Visual Studio .NET instead of doing it directly using paper & pencil. After you have completed the implementation, submit your program file (i.e. FindPrimeNumbers.cpp for this lab) using the electronic “drop-box” in Canvas, **Two weeks** after your lab section is conducted. For details, please refer to the following:

<i>Tuesday sections</i>	<i>by 19:00 on 16 February 2016 (Tuesday)</i>
<i>Wednesday sections</i>	<i>by 19:00 on 17 February 2016 (Wednesday)</i>

You are reminded to double-check your solution to verify everything is correct before submission. You are also required to put your name and student ID at the beginning of your source file.

Important: You are only able to submit your work once.

B. Objective

The objective of this lab is to help you get familiar with pointer declaration and manipulation that you learned in the last two lectures. As pointers are highly related to arrays in C++, more training on arrays will be given so as to reinforce your concepts. Also, you will be continuously introduced to the difference between Java and C++ so as to speed you up your learning process.

The first two part of this lab is meant to be a simple review of pointer and array declaration and manipulation using Microsoft Visual Studio .NET. The last part of this lab is a couple of practical tasks that allow you to use all the techniques introduced (i.e. pointers and arrays).

C. Review

1. What are pointers?

We saw early in the course that variables are essentially storage boxes, in which one can store something – value. These boxes have an address by which the computer can find them, and a name by which the programmer can refer to them, and there are different types of boxes that can hold different amounts of information.

Pointers are variables whose value is the address of another variable.

2. Why do we need them?

There are many reasons. Let's start with two.

- a) First, pointers enable us to create arrays.
- b) Second, pointers enable functions to modify their arguments and to pass back the result.
(i.e. just like reference variables → `void swap2(int& a, int& b);`)

Both of these topics will be discussed in more detail in the next section.

3. How to declare pointers / pointer variables and do initialization?

A typical pointer declaration looks like the following:

- Syntax:

```
// Declare a pointer named <name of pointer variable> in type <type>,  
// which means it can be used to store the address of another variable in type <type>  
  
<type>* <name of pointer variable>; OR  
<type> *<name of pointer variable>;
```

where <type> could be in one of the following types:

Integer data types	short, int, long
Floating point data types	float, double
Character type	char
Boolean type	bool
User-defined type	All other types

- Examples:

```
int* pInt; // A pointer variable named pInt, which can be used to store the
           // address of another variable in type "int"

double* pDouble; // A pointer variable named pDouble, which can be used to store
                  // the address of another variable in type "double"
```

As with typical variables, it is not compulsory to initialize pointers, but if you don't assign their initial values, the content in those variables will be some **garbage value**.

So, we generally initialize newly created pointer variable to **NULL**, which means the pointer currently doesn't yet point to anything. Note: **NULL** is defined in the `iostream` library, therefore we need to put the following preprocessor directive and using statement at the beginning of our program code.

```
#include <iostream>
using namespace std;
```

Extra information: Although there are many types of variables, each taking up different amount of memory, addresses in memory are always the same size, whatever is stored in them. Thus pointers to all types of variables are the same size (usually relatively small).

4. How to obtain address of a variable?

Once we have declared a pointer / pointer variable, the first thing we might do with it is to make it stores the address of other variable.

To obtain the address of a variable, we need to use the "Address of" operator **&**

- Examples:

```
int a = 5; // Declare a typical integer variable named a and store a value 5

int* aPtr = NULL; // Initialize the pointer variable aPtr to NULL

aPtr = &a; // Obtain the address of the typical variable a using address of
           // operator & and assign it to aPtr
```

5. How to use pointer variables?

Pointer variables can be manipulated in several ways.

- Using them to access the content of the typical variables that they point to (i.e. the variables where their addresses are stored)
- Performing **TWO** arithmetic operations: addition (+, ++, +=) and subtraction (-, --, -=) to walkthrough the memory
- Doing comparison: compare the addresses stored in the pointer variables using relational operators (i.e. >, >=, <, <=, ==, !=)

For a) – We need to use dereference operator * to do the job.

- Syntax:

```
// Dereference the pointer variable, i.e. so called “bring us to” the address location
// stored in the pointer variable. If the *<name of pointer variable> is on the left hand
// side, we get back the typical variable. If the *<name of pointer variable> is on the
// right hand side, we get back the value stored in the typical variable

*<name of pointer variable> = <value>;           // left hand side → get back variable
<variable name> = *<name of pointer variable>;    // right hand side → get back value
```

- Examples:

```
int a = 5;

int* aPtr = NULL;
aPtr = &a;
*aPtr = 20; // Bring us to address location stored in aPtr.

           // As *aPtr is on the left hand side, we get back the variable a.
           // So, this statement assign a value 20 to a

int b = *aPtr; // Bring us to address location stored in aPtr
           // As *aPtr now is on the right hand side, we get back the value stored
           // in variable a. So, this statement assign a value 20 to variable "b"

cout << "Value of *aPtr: " << *aPtr << endl; // Display value of *aPtr
```

For b) – Perform arithmetic operations

Examples:

```
int a = 5;

int* aPtr = &a;

cout << "Content in aPtr " << aPtr << endl;

aPtr++;

aPtr += 2;

cout << "Content in aPtr: " << aPtr << endl;

*aPtr = 10;

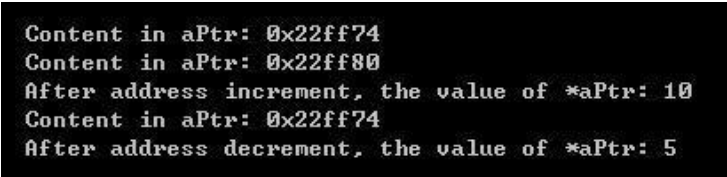
cout << "After address increment, the value of *aPtr: " << *aPtr << endl; aPtr -= 2;

aPtr--;

cout << "Content in aPtr: " << aPtr << endl;

cout << "After address decrement, the value of *aPtr: " << *aPtr << endl;
```

Output:



```
Content in aPtr: 0x22ff74
Content in aPtr: 0x22ff80
After address increment, the value of *aPtr: 10
Content in aPtr: 0x22ff74
After address decrement, the value of *aPtr: 5
```

Address 0x22ff74 (in Hex) = 2293620 (in Dec)

Address 0x22ff80 (in Hex) = 2293632 (in Dec)

From this example, we should be able to see that the pointer increment by statements:

`aPtr++; aPtr += 2;`

add 12 (= 2293632 – 2293620, i.e. add 12 bytes) to aPtr.

This is due to the fact that `aPtr++` and `aPtr += 2` **moves 3 int locations** downward as an int variable occupied 4 bytes in VC++.

Question:

What happen if the following code segment is executed?

```
int* myPtr = NULL;

*myPtr = 10;
```

Answer:

Run-time error occurs. Since the above code tries to access the memory location 0 (NULL means address 0) that is not supposed to be used to store anything.

For c) – Perform relational operations

Examples:

```
int a = 5;
int b = 10;

int* aPtr = &a;
int* bPtr = &b;

if( aPtr < bPtr )
    cout << "Address in aPtr is before the address in bPtr" << endl;
else
    cout << "Address in aPtr is after the address in bPtr" << endl;
```

Static Array and Dynamic Array Allocation

As mentioned in the last lab, C++ offers the flexibility to declare an array *using new* or *not using new* keyword.

The following shows an example of array declaration **without** using new keyword. We call this “**static array allocation**”.

```
int a[5];    // Create an array of integers with 5 elements
```

Important:

In this example, variable a is actually a **constant pointer**, i.e. a pointer variable in which the content (address) cannot be changed. Variable “a” stores the address of the first element, i.e. a[0].

Constant pointer can be declared as follows:

```
int a = 20;
int b = 10;
int* const c = &a; // Here c is a constant pointer, i.e. the address in c CANNOT be changed
c = &b;           // This one is WRONG
const int* d = &b; // Here, the address in d CAN be changed, but the address location pointed
                  // by d CANNOT be changed
*d = 20;         // This one is WRONG
int e[10];       // Declare a static array e, e is actually a constant pointer
*e = 50;         // Assign 50 to the first location of the array
e = &b;          // Error: the content of e cannot be changed, since it is a constant pointer
```

In this lab, we only focus on declaring arrays *using new*. We call this “**dynamic array allocation**”.

- Syntax:

```
// Declare an array named <name of variable> of size <size> in type <type> using new  
<type>* <name of variable> = new <type>[<size>];
```

- Examples:

```
int* arr1 = new int[10]; // An array arr1 with 10 elements  
  
int n;  
  
cin >> n;  
  
int* arr2 = new int[n]; // An array arr2 with size depending on the user input
```

Question:

Why creating an array using new is important in C++?

Answer:

This is only way in C++, which can be used to create array in variable size. This is crucial as the size of array may not able to be determined during compilation time.

e.g. Open and read an image. Its size is only known during the runtime.

Question:

Apart from the dynamic size, is there any difference between creation of array using new and not using new?

Answer:

Yes, the memory allocated to array **without using new** will be **returned to operating system automatically** when it is no longer usable. But the array memory locations **allocated using new WILL NOT be returned to operating system automatically**. It has to be done by the programmer using the **delete** keyword as follows:

Syntax:

// Return allocated memory back to operating system using delete keyword.

// If DYNAMIC ARRAY is allocated, [] has to be put together with the delete keyword.

delete [] <name of pointer variable for array>;

- Example of new and delete:

```
int n;
cin >> n;

// An array arr with size depending on the user input
int* arr = new int[n];
// ...

delete [ ] arr;
```

More on Declaration and Definition of User-defined Function

Recall, we went through an example of swapping two values between two variables using operator & in the last lab. In this lab, we try to achieve the same thing using pointers.

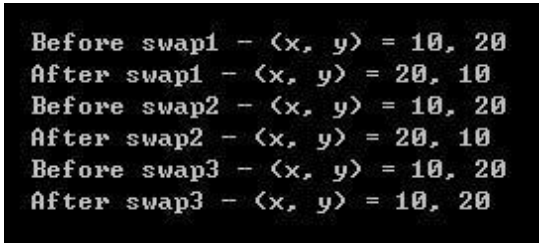
- Examples:

```
#include <iostream>
using namespace std;

// Swapping two integers

void swap1( int& a, int& b ) {
    int temp = a;
    a = b;
    b = temp;
}

void swap2( int* a, int* b ) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```



```
Before swap1 - <x, y> = 10, 20
After swap1 - <x, y> = 20, 10
Before swap2 - <x, y> = 10, 20
After swap2 - <x, y> = 20, 10
Before swap3 - <x, y> = 10, 20
After swap3 - <x, y> = 10, 20
```



```

void swap3( int* a, int* b ) {
    int* temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;

    cout << "Before swap1 - (x, y) = " << x << ", " << y << endl;
    swap1( x, y );

    cout << "After swap1 - (x, y) = " << x << ", " << y << endl;
    x = 10, y = 20;

    cout << "Before swap2 - (x, y) = " << x << ", " << y << endl;
    swap2( &x, &y );

    cout << "After swap2 - (x, y) = " << x << ", " << y << endl;
    x = 10, y = 20;

    cout << "Before swap3 - (x, y) = " << x << ", " << y << endl;
    swap3( &x, &y );

    cout << "After swap3 - (x, y) = " << x << ", " << y << endl;
    system("pause");
    return 0;
}

```

Question:

Which swapping function(s), swap1, swap2 or swap3 does (do) the job correctly?

Answer:

swap1 and swap2 are the correct ones to do the swapping of two integers.

D. Find Prime Numbers

In this part, you are required to:

- i) Create a New Project and give your project the name Lab4b.
- ii) Add a source file to your project, called FindPrimeNumbers.cpp
- iii) Write a C++ program that uses an array of n elements (where n is a user input) to determine and display the prime numbers between 2 and n using the method described below. Ignore array elements 0 and 1.

A prime number is any integer greater than one that is divisible only by itself and 1. A list of prime numbers can be found by the following method.

- a) Create a primitive type ***bool*** array with all elements initialized to true. Array elements with prime indices will remain true. All other array elements will eventually be set to false.
- b) Starting with array index 2, determine whether a given element is true. If so, loop through the remaining of the array and set to false every element whose index is a multiple of the index for the element with value true.
- c) Then continue the process with the next element with value true.

For array index 2, all elements beyond element 2 in the array that have indices which are multiples of 2 (indices 4, 6, 8, 10, etc.) will be set to false; for array index 3, all elements beyond element 3 in the array that have indices which are multiples of 3 (indices 6, 9, 12, 15, etc.) will be set to false; and so on.

- d) When this process completes, the array elements that are still true indicate that the index is a prime number. These indices can be displayed.

Multiple sessions of sample screen display when the method is called are given below:

```
Enter the value of n: 10
Prime numbers: 2, 3, 5, 7
4 primes found.
```

```
Enter the value of n: 60
Prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59
17 primes found.
```

Enter the value of n: 100

Prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
61, 67, 71, 73, 79, 83, 89, 97

25 primes found.

The input which is underlined is the user's input to a question. (Note: you don't have to repeatedly execute your program code using loop for this assignment)

- iv) Compile your program and test it by executing your program.

Marking Scheme:

Graded items	Weighting
1. Correctness of program (i.e. whether your code is implemented in a way according to the requirements as specified.)	60%
2. Indentation	30%
3. Documentation (with reasonable amount of comments embedded in the code to enhance the readability.)	10%
	100%

Program Submission Checklist

Before submitting your work, please check the following items to see you have done a decent job.

Items to be checked

☑ / ☒

1. Did I put my name and student ID at the beginning of all the source files? ☐
2. Did I put reasonable amount of comments to describe my program? ☐
3. Are they all in .cpp extension and named according to the specification? ☐
4. Have I checked that all the submitted code are compliable and run without any errors? ☐
5. Did I zip my source files using Winzip / zip provided by Microsoft Windows? Also, did I check the zip file and see if it could be opened? ☐
(Only applicable if the work has to be submitted in zip format.)
6. Did I submit my lab assignment to Canvas? ☐

-End-