|  |
| --- |
| **Programming Fundamentals** |
| **(CL214)** |
| **LABORATORY MANUAL** |
| **Spring 2021** |
| **C:\Users\Aamer\Desktop\nu-new.png**  **LAB 06** |
| **Pointers to Structure** |
| **Engr. Ibrar Khan**  **Engr. Sana Saleh**  **Muhammad Adeel Akhtar 20i-1025 B** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | \_\_\_\_\_\_\_\_\_\_ | | | \_\_\_ |
| STUDENT NAME | | ROLL NO | | | SEC |
|  | | | | | |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | | | |
| LAB ENGINEER SIGNATURE & DATE | | | | | |
| **MARKS AWARDED:** \_\_\_**/10** | | | | | |
|  | | | | | |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (NUCES), ISLAMABAD** | | | | | |
|  | | | | | |
| Prepared by: | Engr. Sana Saleh | | Date: | 18 Sept, 2019 | |
| Verified by: | Engr. Sana Hassan | | Date: | 18 Sept, 2019 | |

|  |  |
| --- | --- |
| **LAB 06** | ***Pointers to Structures*** |

**Lab Objectives:**

1. To learn and revise pointers basic.
2. Pass pointers to functions.
3. To learn about pointers to structures.

**Software Required:**

* Dev C++

**Introduction:**

**Pointers**

### What is a Pointer?

A pointer is a variable that stores a memory address.  Pointers are used to store the addresses of other variables or memory items.  Pointers are very useful for another type of parameter passing, usually referred to as **Pass by Address**.  Pointers are essential for dynamic memory allocation.

### Pointer types and Type Casting:

Although all pointers are addresses (and therefore represented similarly in data storage), we want the type of the pointer to indicate what is being pointed to.  Therefore, C++ treats pointers to different types AS different types themselves.

int \* ip;

char \* cp;

double \* dp;

These three pointer variables (ip, dp, cp) are all considered to have different types, so assignment between any of them is illegal.  The automatic type coercions that work on regular numerical data types **do not apply**:

ip = dp; // ILLEGAL

dp = cp; // ILLEGAL

ip = cp; // ILLEGAL

As with other data types, you can always force a coercion by re-casting the data type.  Be careful that you know what you are doing, though, if you do this!

ip = static\_cast<int \*>(dp);

1. **The "address of" operator:**

There is another use of the & operator.  When it is used in a declaration, it creates a **reference**variable. When the & is used in regular statements (not reference declarations), the operator means "address of".

**Example**:

int n; // integer

cout << &n; // this prints out &n, the "address of n".

int \* p; // pointer to an integer

p = &n; // assigns the "address of n" to the pointer p

1. **Pass By Address:**

If you declare a formal parameter of a function as a pointer type, you are passing that parameter by its address.  The pointer is copied, but not the data it points to.  So, Pass By Address offers another method of allowing us to change the original argument of a function (like with Pass By Reference).  Don't pass in the argument itself -- just pass in its address.

**Example:**

int main()

{

int num = 4;

cout << "Original = " << num << '\n';

SquareByAddress(&num);

cout << "New value = " << num << '\n';

}

void SquareByAddress(int \* n)

{

\*n = (\*n) \* (\*n);

}

* + - 1. **Pointers and Arrays:**

1. **Introduction:**

When you declare an array normally, you get a pointer for free.  The **name** of the array acts as a pointer to the first element of the array.

int list[10]; // the variable list is a pointer

// to the first integer in the array

int \* p; // p is a pointer. It has the same type as list.

p = list; // legal assignment. Both pointers to ints.

In the above code, the address stored in list has been assigned to p.  Now both pointers point to the first element of the array.  Now, we could actually use p as the name of the array!

list[3] = 10;

p[4] = 5;

cout << list[6];

cout << p[6];

In the above array example, we referred to an array item with p[6].  We could also say \*(p+6).  When you add to a pointer, you do not add the literal number.  You add that number of units, where a unit is the type being pointed to.  For instance, p + 6 in the above example means to move the pointer forward 6 integer addresses.  Then we can dereference it to get the data  \*(p + 6).Most often, pointer arithmetic is used in conjunction with arrays.

**Example:**

Suppose ptr is a pointer to an integer, and ptr stores the address 1000.  Then the expression (ptr + 5) does not give 1005 (1000+5).  Instead, the pointer is moved 5 integers (ptr + (5 \* size-of-an-int)).  So, if we have 4-byte integers, (ptr+5) is 1020 (1000 + 5\*4).

### Pass By Address with arrays:

The fact that an array's name is a pointer allows easy passing of arrays in and out of functions.  When we pass the array in by its name, we are passing the **address** of the first array element. When an array is passed into a function (by its name), any changes made to the array elements do affect the original array, since only the array **address** is copied (not the array elements themselves).

void Swap(int \* list, int a, int b)

{

int temp = list[a];

list[a] = list[b];

list[b] = temp;

}

This Swap function allows an array to be passed in by its name only.  The pointer is copied but not the entire array.  So, when we swap the array elements, the changes are done on the original array.  Here is an example of the call from outside the function:

int numList[5] = {2, 4, 6, 8, 10};

Swap(numList, 1, 4); // swaps items 1 and 4

An array passed into a function is always passed by address, since the array's name IS a variable that stores its address (i.e. a pointer).Pass-by-address can be done in returns as well -- we can return the address of an array.

int \* ChooseList(int \* list1, int \* list2)

{

if (list1[0] < list2[0])

return list1;

else

return list2; // returns a copy of the address of the array

}

And an example usage of this function:

int numbers[5] = {1,2,3,4,5};

int numList[3] = {3,5,7};

int \* p;

p = ChooseList(numbers, numList);

* + - 1. **Pointers & Structures:**

Just like other pointers, the structure pointers are declared by placing asterisk (∗) in front of a structure pointer's name. It takes the following general form :

struct-name \*struct-pointer;

where struct-name is the name of an already defined structure and struct-pointer is the pointer to this structure. For example, to declare dt-ptr as a pointer to already defined structure date, we shall write

date \*dt-ptr;

sing structure pointers, the members of structures are accessed using arrow operator ->. To refer to the structure members using -> operator, it is written as

struct-pointer -> structure-member

***static* Keyword*:***

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime.

**Example:**

|  |
| --- |
| #include<iostream>  using namespace std;  struct date  {  short int dd, mm, yy;  };  void main()  {  date join\_date = { 19, 12, 2006 };  date \*date\_ptr;  date\_ptr = &join\_date;  cout << "Printing the structure elements using the structure variable\n";  cout << "dd = " << join\_date.dd << ", mm = " << join\_date.mm << ", yy = " << join\_date.yy << "\n";  cout << "\nPrinting the structure elements using the structure pointer\n";  cout << "dd = " << date\_ptr->dd << ", mm = " << date\_ptr->mm << ", yy = " << date\_ptr->yy << "\n";  } |

# Dynamic Memory Allocation

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space.

The answer is dynamic memory, for which C++ integrates the operators *new* and *delete*.

new

new[]

In order to request dynamic memory we use the operator *new*. *new* is followed by a data type specifier and -if a sequence of more than one element is required- the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

*pointer = new type*

*pointer = new type [NumberOfElements]*

The first expression is used to allocate memory to contain one single element of type *type*. The second one is used to assign a block (an array) of elements of type *type*, where *NumberOfElements* is an integer value representing the amount of these. For example:

int \* bobby;

bobby = new int [5];

In this case, the system dynamically assigns space for five elements of type *int* and returns a pointer to the first element of the sequence, which is assigned to bobby. Therefore, now, bobby points to a valid block of memory with space for five elements of type *int*.

The first element pointed by bobby can be accessed either with the expression bobby[0] or the expression \*bobby. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with bobby[1] or \*(bobby+1) and so on...

You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

Operator *delete* and *delete[]*

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator *delete*, whose format is:

delete pointer;

delete [] pointer;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements. The value passed as argument to *delete* must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

// Example program for dynamic memory allocation

# include <iostream>

using namespace std;

int main ()

{

int i,n;

int \* p;

cout << "How many numbers would you like to type ? ";

cin >> i;

p = new int[i];

for (n=0; n<i; n++)

{

cout << " Enter number : ";

cin >> p[n];

}

cout << "You have entered : ";

for (n=0; n<i; n++)

cout << p[n] << ", ";

delete [] p;

return 0;

}

**Example:**

The following program asks the user to enter the number of points dynamically. This data is then printed on the console using output() function.

#include<iostream>

using namespace std;

int n;

struct point{int x,y; };

point\* input()

{

cout<<"Enter number of points you want to enter\n";

cin>>n;

point\* p=new point[n];

for(int x=0;x<n;x++)

{

cout<<"Enter x and y coordinate of "<<n<<" point\n";

cin>>p->x;

cin>>p->y;

}

return p;

}

void print(point \*p)

{

cout<<"Point\t" <<" x coordinate\t y coordiante\n";

for(int x=0;x<n;x++)

{

cout<<x<<"\t"<<p->x<<"\t"<<p->y<<endl;

}

}

int main()

{

point\* p= input();

print(p);

}

**Practice Problem:**

* 1. Write a program that can double the two numbers entered by user. Use functions for the implementation of this program. Print value of variables in function body as well as in main(), immediately after function call. Ask user whether he want to pass arguments by:
     1. value.
     2. reference.
     3. Passing address of pointers.
     4. Passing Array (Using pointer notation)

Carefully use pointer notation to access each element of array.

#include<iostream>

using namespace std;

void doub(int,int);

void doub1(int&,int&);

void doub2(int \*p1,int \*p2);

void doub3(int ptr[]);

int main()

{

int \*p1,\*p2;

int x,y,k,o;

cout<<"enter first number"<<endl;

cin>>x;

k=x;

int n;

n=k;

cout<<"enter second number"<<endl;

cin>>y;

o=y;

int m;

m=o;

doub(x,y);

cout<<"In pass by value inside main first double value is"<<endl<<x<<endl;

cout<<"In pass by value inside main second double value is"<<endl<<y<<endl;

doub1(x,y);

cout<<"In pass by reference inside main first double value is"<<endl<<x<<endl;

cout<<"In pass by reference inside main second double value is"<<endl<<y<<endl;

doub2(&k,&o);

cout<<"In pass by pointer inside main first double value is"<<endl<<x<<endl;

cout<<"In pass by pointer inside main second double value is"<<endl<<y<<endl;

int a[2]={n,m};

int \*ptr;

ptr=a;

doub3(ptr);

cout<<"In pass by array inside main first double value is"<<endl<<ptr[0]<<endl;

cout<<"In pass by pointer inside main second double value is"<<endl<<ptr[1]<<endl;

}

void doub(int x,int y)

{

int z,a;

x\*=2;

y\*=2;

cout<<"Inside pass by value fuction first double value is"<<endl<<x<<endl;

cout<<"Inside pass by value fuction second double value is"<<endl<<y<<endl;

}

void doub1(int &x,int &y)

{

int z,a;

x\*=2;

y\*=2;

cout<<"Inside pass by reference fuction first double value is"<<endl<<x<<endl;

cout<<"Inside pass by reference fuction second double value is"<<endl<<y<<endl;

}

void doub2(int \*p1,int \*p2)

{

int z;

\*p1=2\*(\*p1);

cout<<"Inside pass by pointer adress fuction first double value is"<<endl;

cout<<\*p1<<endl;

\*p2=2\*(\*p2);

cout<<"Inside pass by pointer adress fuction first double value is"<<endl;

cout<<\*p2<<endl;

}

void doub3(int ptr[])

{

ptr[0]\*=2;

ptr[1]\*=2;

cout<<"Inside pass by array fuction first double value is"<<endl;

cout<<ptr[0]<<endl;

cout<<"Inside pass by array fuction second double value is"<<endl;

cout<<ptr[1]<<endl;

}

* 1. Type-Casting:

Dry Run this code. Write output values you are expecting. For garbage value write ”GB”.

|  |  |
| --- | --- |
| **Code**  void main()  {  int a[10] = { 0,1, 2, 3, 4, 5, 6, 7, 8, 9 };  int \*p = a;  char \*b=(char\*)p;  cout << \*(int\*)(b + 0) << endl;  cout << \*(int\*)(b + 1) << endl;  cout << \*(int\*)(b + 4)<<endl;  cout << \*(int\*)(b + 32) << endl;  cout << \*(int\*)(b + 36) << endl;  cout << \*(p + 1) << endl;  cout << \*(p + 5) << endl;  cout << \*(p + 9) << endl;  cout << \*(p + 32) << endl;  double \*x;  x = (double\*)p;  cout << \*(int\*)(x + 0) << endl;  cout << \*(int\*)(x + 1) << endl;  cout << \*(int\*)(x + 4) << endl;  cout << \*(int\*)(x + 9) << endl;  } | **Dry Run**  0  16777216(GB)  1  8  9  1  5  9  0  0  2  8  4199400(GB) |

1. Student Marks in Programming Fundamental course
2. Ask user to enter number of students enrolled in PF and entered each student marks. (Use Dynamic memory allocation). Implement it using any one of the function prototypes given below:

**void input (int \*p) OR int\* input()**

1. To know these student’s performance in last year programming course i.e. ITC, store each student grade. (Use Dynamic memory allocation). Implement it using any one of the function prototypes given below:

**void grade (char \*G) OR char\* grade()**

1. Create a structure student, that has three members element i.e. name, PF marks and ITC grade. Using dynamic memory allocation, create an array of student. Copy data of above stored information in each structure. Access each member using (->) operator.

**Student\* copy(int \*,char \*) OR void copy(Student\*,int\*,char\*)**

1. Sort Student data in ascending order according to marks obtained in ITC course using function prototype given below:

**void sort(Student\*, int)**

1. Print each student data on console.

**void print (Student\* , int number)**

1. Delete all pointers.