**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **23-06-2020** | **Name:** | **Karthik J** |
| **Course:** | C++ Tutorials - Solo Learn | **USN:** | **4AL16EC030** |
| **Topic:** | Data types, Arrays, Pointers | **Semester & Section:** | **8TH A** |
| **GitHub Repository:** | Karthik-J |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image Section**    Data Types, Arrays, Pointers  Introduction to Data Types Data Types The operating system allocates memory and selects what will be stored in the reserved memory based on the variable's **data type**.The data type defines the proper use of an identifier, what kind of data can be stored, and which types of operations can be performed.Expressions The examples below show legal and illegal C++ expressions.55+15 // **legal** C++ expression //Both operands of the + operator are integers  55 + "John" // **illegal** // The + operator is not defined for integer and string Numeric Data Types Numeric data types include: **Integers** (whole numbers), such as -7, 42. **Floating point** numbers, such as 3.14, -42.67.Strings & Characters A **string** is composed of numbers, characters, or symbols. String literals are placed in **double quotation** marks; some examples are "Hello", "My name is David", and similar.  **Characters** are single letters or symbols, and must be enclosed between **single quotes**, like 'a', 'b', etc. Booleans  The Boolean data type returns just two possible values: **true** (1) and **false** (0).Integers The **integer** type holds non-fractional numbers, which can be positive or negative. Examples of integers would include 42, -42, and similar numbers. Integers Use the **int** keyword to define the integer data type.**int**  a = 42; Several of the basic types, including integers, can be modified using one or more of these type **modifiers**:  **signed**: A signed integer can hold both negative and positive numbers. **unsigned**: An unsigned integer can hold only positive values. **short**: Half of the default size. **long**: Twice the default size. Floating Point Numbers A **floating point** type variable can hold a real number, such as 420.0, -3.33, or 0.03325.  The words floating point refer to the fact that a varying number of digits can appear before and after the decimal point. You could say that the decimal has the ability to "**float**".   There are three different floating point data types: **float**, **double**, and **long double**.  In most modern architectures, a **float** is 4 bytes, a **double** is 8, and a **long double** can be equivalent to a double (8 bytes), or 16 bytes. Arrays An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.  That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index. Initializing arrays By default, regular arrays of local scope (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.  But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:   |  |  |  | | --- | --- | --- | |  | int foo [5] = { 16, 2, 77, 40, 12071 }; |  |  Accessing the values of an array The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:  name[index] Multidimensional arrays Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type. Arrays as parameters At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.  To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:   |  |  |  | | --- | --- | --- | |  | void procedure (int arg[]) |  |   This function accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:   |  |  |  | | --- | --- | --- | |  | int myarray [40]; |  |   it would be enough to write a call like this:   |  |  |  | | --- | --- | --- | |  | procedure (myarray); |  |  Pointers In earlier chapters, variables have been explained as locations in the computer's memory which can be accessed by their identifier (their name). This way, the program does not need to care about the physical address of the data in memory; it simply uses the identifier whenever it needs to refer to the variable.  For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address. These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.  This way, each cell can be easily located in the memory by means of its unique address. For example, the memory cell with the address 1776 always follows immediately after the cell with address 1775 and precedes the one with 1777, and is exactly one thousand cells after 776 and exactly one thousand cells before 2776.  When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address). Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored. Fortunately, that task is left to the environment where the program is run - generally, an operating system that decides the particular memory locations on runtime. However, it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it. Address-of operator (&) The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator. For example:   |  |  |  | | --- | --- | --- | |  | foo = &myvar; |  |   This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the address-of operator (&), we are no longer assigning the content of the variable itself to foo, but its address.  The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that myvar is placed during runtime in the memory address 1776.  In this case, consider the following code fragment:   |  |  |  | | --- | --- | --- | | 1 2 3 | myvar = 25;  foo = &myvar;  bar = myvar; |  |  Dereference operator (\*) As just seen, a variable which stores the address of another variable is called a pointer. Pointers are said to "point to" the variable whose address they store.  An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the dereference operator (\*). The operator itself can be read as "value pointed to by".  Therefore, following with the values of the previous example, the following statement:   |  |  |  | | --- | --- | --- | |  | baz = \*foo; |  |  Declaring pointers Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it points to a char than when it points to an int or a float. Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.  The declaration of pointers follows this syntax:  type \* name;   where type is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:   |  |  |  | | --- | --- | --- | | 1 2 3 | int \* number;  char \* character;  double \* decimals; |  |   These are three declarations of pointers. Each one is intended to point to a different data type, but, in fact, all of them are pointers and all of them are likely going to occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the program runs). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an int, the second one to a char, and the last one to a double. Therefore, although these three example variables are all of them pointers, they actually have different types: int\*, char\*, and double\* respectively, depending on the type they point to.  Note that the asterisk (\*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator seen a bit earlier, but which is also written with an asterisk (\*). They are simply two different things represented with the same sign. Pointers and arrays The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:   |  |  |  | | --- | --- | --- | | 1 2 | int myarray [20];  int \* mypointer; |  |   The following assignment operation would be valid:   |  |  |  | | --- | --- | --- | |  | mypointer = myarray; |  |   After that, mypointer and myarray would be equivalent and would have very similar properties. The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would not be valid:   |  |  |  | | --- | --- | --- | |  | myarray = mypointer; |  |   #include <iostream>  using namespace std;  int main ()  {  int numbers[5];  int \* p;  p = numbers; \*p = 10;  p++; \*p = 20;  p = &numbers[2]; \*p = 30;  p = numbers + 3; \*p = 40;  p = numbers; \*(p+4) = 50;  for (int n=0; n<5; n++)  cout << numbers[n] << ", ";  return 0;  } Pointer initialization Pointers can be initialized to point to specific locations at the very moment they are defined:   |  |  |  | | --- | --- | --- | | 1 2 | int myvar;  int \* myptr = &myvar; |  |   The resulting state of variables after this code is the same as after:   |  |  |  | | --- | --- | --- | | 1 2 3 | int myvar;  int \* myptr;  myptr = &myvar; |  |   When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., \*myptr). Therefore, the code above shall not be confused with:   |  |  |  | | --- | --- | --- | | 1 2 3 | int myvar;  int \* myptr;  \*myptr = &myvar; |  |   Which anyway would not make much sense (and is not valid code).  The asterisk (\*) in the pointer declaration (line 2) only indicates that it is a pointer, it is not the dereference operator (as in line 3). Both things just happen to use the same sign: \*. As always, spaces are not relevant, and never change the meaning of an expression.  Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array):   |  |  |  | | --- | --- | --- | | 1 2 3 | int myvar;  int \*foo = &myvar;  int \*bar = foo; |  |    Dynamic memory In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete. Operators new and new[] Dynamic memory is allocated using 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 syntax is:   pointer = new type pointer = new type [number\_of\_elements]  The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number\_of\_elements is an integer value representing the amount of these. For example:   |  |  |  | | --- | --- | --- | | 1 2 | int \* foo;  foo = new int [5]; |  |    In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid block of memory with space for five elements of type int.  Operators delete and delete[] In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:   |  |  |  | | --- | --- | --- | | 1 2 | delete pointer;  delete[] pointer; |  |   The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).  The value passed as argument to delete shall 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).  #include <iostream>  #include <new>  using namespace std;  int main ()  {  int i,n;  int \* p;  cout << "How many numbers would you like to type? ";  cin >> i;  p= new (nothrow) int[i];  if (p == nullptr)  cout << "Error: memory could not be allocated";  else  {  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;  } Dynamic memory in C C++ integrates the operators new and delete for allocating dynamic memory. But these were not available in the C language; instead, it used a library solution, with the functions [malloc](http://www.cplusplus.com/malloc), [calloc](http://www.cplusplus.com/calloc), [realloc](http://www.cplusplus.com/realloc) and [free](http://www.cplusplus.com/free), defined in the header [<cstdlib>](http://www.cplusplus.com/%3ccstdlib%3e) (known as <stdlib.h> in C). The functions are also available in C++ and can also be used to allocate and deallocate dynamic memory.  Note, though, that the memory blocks allocated by these functions are not necessarily compatible with those returned by new, so they should not be mixed; each one should be handled with its own set of functions or operators. |