Arrays,String,Pointer, Dynamic memory

C++ Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store:

```
string cars[4];
```

We have now declared a variable that holds an array of four strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of three integers, you could write:

```
int myNum[3] = {10, 20, 30};
```

Access the Elements of an Array

You access an array element by referring to the index number inside square brackets `[]`.

This statement accesses the value of the **first element** in **cars**:

```
#include <iostream>

#include <string>

using namespace std;


int main() {

  string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
  cout << cars[0];
  return 0;
}
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

```
cars[0] = "Opel";
```

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

int main() {
  string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
  cars[0] = "Swift";
  cout << cars[0];
  return 0;
}
```

Loop Through an Array

You can loop through the array elements with the `for` loop.

The following example outputs all elements in the **cars** array:

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

```
int main() {
  string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
  for (int i = 0; i < 5; i++) {
    cout << cars[i] << "\n";
  }
  return 0;
}
```

This example outputs the index of each element together with its value:

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

int main() {
  string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
  for (int i = 0; i < 5; i++) {
    cout << i << " = " << cars[i] << "\n";
  }
  return 0;
}
```

And this example shows how to loop through an array of integers:

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

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  for (int i = 0; i < 5; i++) {
    cout << myNumbers[i] << "\n";
  }
```

```
  return 0;
}
```

Get the Size of an Array

To get the size of an array, you can use the `sizeof()` operator:

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

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  cout << sizeof(myNumbers);
  return 0;
}
```

Why did the result show `20` instead of `5`, when the array contains 5 elements?

It is because the `sizeof()` operator returns the size of a type in **bytes**.

You learned from the Data Types chapter that an `int` type is usually 4 bytes, so from the example above, *4 x 5 (4 bytes x 5 elements)* = ***20 bytes***.

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

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  int getArrayLength = sizeof(myNumbers) / sizeof(int);
  cout << getArrayLength;
  return 0;
}
```

Loop Through an Array with sizeof()

In the Arrays and Loops Chapter, we wrote the size of the array in the loop condition (`i < 5`). This is not ideal, since it will only work for arrays of a specified size.

However, by using the `sizeof()` approach from the example above, we can now make loops that work for arrays of any size, which is more sustainable.

Instead of writing:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
  cout << myNumbers[i] << "\n";
}
```

It is better to write:

```
#include <iostream>

using namespace std;


int main() {

  int myNumbers[5] = {10, 20, 30, 40, 50};

  for (int i = 0; i < sizeof(myNumbers) / sizeof(int); i++) {

    cout << myNumbers[i] << "\n";

  }

  return 0;

}
```

C++ Strings

Strings are used for storing text.

A `string` variable contains a collection of characters surrounded by double quotes:

Example

Create a variable of type `string` and assign it a value:

```
string greeting = "Hello";
```

To use strings, you must include an additional header file in the source code, the `<string>` library:

Example

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string greeting = "Hello";
  cout << greeting;
  return 0;
}
```

String Concatenation

The + operator can be used between strings to add them together to make a new string. This is called **concatenation**:

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

int main () {
  string firstName = "karan ";
  string lastName = “johar";
  string fullName = firstName + lastName;
  cout << fullName;
  return 0;
}
```

In the example above, we added a space after firstName to create a space between karan and johar on output. However, you could also add a space with quotes (" " or ' '):

Example

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

int main () {
  string firstName = "John";
  string lastName = "Doe";
  string fullName = firstName + "  " + lastName;
  cout << fullName;
  return 0;
}
```

Append

A string in C++ is actually an object, which contain functions that can perform certain operations on strings. For example, you can also concatenate strings with the `append()` function:

Example

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

int main () {
  string firstName = "sushant";
  string lastName = "rajput";
  string fullName = firstName.append(lastName);
  cout << fullName;
  return 0;
}
```

Adding Numbers and Strings

WARNING!

C++ uses the + operator for both **addition** and **concatenation**.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

Example

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

int main () {
  int x = 10;
  int y = 20;
  int z = x + y;
  cout << z;
  return 0;
}
```

If you add two strings, the result will be a string concatenation:

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

int main () {
  string x = "10";
  string y = "20";
  string z = x + y;
  cout << z;
  return 0;
}
```

If you try to add a number to a string, an error occurs:

Example

```
string x = "10";
int y = 20;
string z = x + y;
```

String Length

To get the length of a string, use the `length()` function:

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

int main() {
  string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << "The length of the txt string is: " << txt.length();
  return 0;
}
```

Tip: You might see some C++ programs that use the `size()` function to get the length of a string. This is just an alias of `length()`. It is completely up to you if you want to use `length()` or `size()`:

Example

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

int main() {
  string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  cout << "The length of the txt string is: " << txt.size();
  return 0;
}
```

Access Strings

You can access the characters in a string by referring to its index number inside square brackets `[]`.

This example prints the **first character** in **myString**:

Example

```
#include <iostream>

#include <string>

using namespace std;


int main() {

  string myString = "Rohit";

  cout << myString[0];

  return 0;

}
```

Note: String indexes start with 0: [0] is the first character. [1] is the second character, etc.

This example prints the **second character** in **myString**:

Example

```
#include <iostream>

#include <string>

using namespace std;


int main() {

  string myString = "Sharma";

  cout << myString[1];

  return 0;

}
```

Change String Characters

To change the value of a specific character in a string, refer to the index number, and use single quotes:

Example

```
#include <iostream>

#include <string>

using namespace std;


int main() {

  string myString = "ROHIT";

  myString[0] = 'J';

  cout << myString;

  return 0;

}
```

User Input Strings

It is possible to use the extraction operator `>>` on `cin` to display a string entered by a user:

Example

```
string firstName;
cout << "Type your first name: ";
cin >> firstName; // get user input from the keyboard
cout << "Your name is: " << firstName;

// Type your first name: John
// Your name is: John
```

However, `cin` considers a space (whitespace, tabs, etc) as a terminating character, which means that it can only display a single word (even if you type many words):

Example

```
string fullName;
cout << "Type your full name: ";
cin >> fullName;
cout << "Your name is: " << fullName;

// Type your full name: rohit sharma

// Your name is rohit sharma
```

From the example above, you would expect the program to print "John Doe", but it only prints "John".

That's why, when working with strings, we often use the `getline()` function to read a line of text. It takes `cin` as the first parameter, and the string variable as second:

Example

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


int main() {
  string fullName;
  cout << "Type your full name: ";
  getline (cin, fullName);
  cout << "Your name is: " << fullName;
  return 0;
}
```

Creating Pointers

that we can get the **memory address** of a variable by using the `&` operator:

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

```
int main() {
  string food = "Pizza";


  cout << food << "\n";
  cout << &food << "\n";
  return 0;
}
```

A **pointer** however, is a variable that **stores the memory address as its value**.

A pointer variable points to a data type (like `int` or `string`) of the same type, and is created with the `*` operator. The address of the variable you're working with is assigned to the pointer:

Example

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


int main() {
  string food = "Pizza";  // A string variable
  string* ptr = &food;  // A pointer variable that stores the address
of food


  // Output the value of food
  cout << food << "\n";


  // Output the memory address of food
  cout << &food << "\n";


  // Output the memory address of food with the pointer
  cout << ptr << "\n";
  return 0;
```

```
}
```

Example explained

Create a pointer variable with the name `ptr`, that **points to** a `string` variable, by using the asterisk sign `*` (`string* ptr`). Note that the type of the pointer has to match the type of the variable you're working with.

Use the `&` operator to store the memory address of the variable called `food`, and assign it to the pointer.

Now, `ptr` holds the value of `food`'s memory address.

Tip: There are three ways to declare pointer variables, but the first way is preferred:

```
string* mystring; // Preferred
string *mystring;
string * mystring;
```

Get Memory Address and Value

In the example from the previous page, we used the pointer variable to get the memory address of a variable (used together with the `&` **reference** operator). However, you can also use the pointer to get the value of the variable, by using the `*` operator (the **dereference** operator):

Example

```
#include <iostream>

#include <string>

using namespace std;


int main() {

  string food = "Pizza";  // Variable declaration

  string* ptr = &food;    // Pointer declaration


  // Reference: Output the memory address of food with the pointer

  cout << ptr << "\n";


  // Dereference: Output the value of food with the pointer
```

```
  cout << *ptr << "\n";

  return 0;

}
```

Note that the * sign can be confusing here, as it does two different things in our code:

- When used in declaration (string* ptr), it creates a **pointer variable**.
- When not used in declaration, it act as a **dereference operator**.

Modify the Pointer Value

You can also change the pointer's value. But note that this will also change the value of the original variable:

Example

```
#include <iostream>

#include <string>

using namespace std;


int main() {

  string food = "Pizza";

  string* ptr = &food;


  // Output the value of food

  cout << food << "\n";


  // Output the memory address of food

  cout << &food << "\n";


  // Access the memory address of food and output its value

  cout << *ptr << "\n";


  // Change the value of the pointer
```

```
  *ptr = "Hamburger";

  // Output the new value of the pointer
  cout << *ptr << "\n";

  // Output the new value of the food variable
  cout << food << "\n";
  return 0;
}
```

Dynamic memory

C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.

In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.

In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.

We can allocate and then deallocate memory dynamically using the `new` and `delete` operators respectively.

C++ new Operator

The `new` operator allocates memory to a variable. For example,

```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// using the new keyword
pointVar = new int;
```

```
// assign value to allocated memory
*pointVar = 45;
```

Here, we have dynamically allocated memory for an `int` variable using the `new` operator.

Notice that we have used the pointer `pointVar` to allocate the memory dynamically. This is because the `new` operator returns the address of the memory location.

In the case of an array, the `new` operator returns the address of the first element of the array.

From the example above, we can see that the syntax for using the `new` operator is

```
pointerVariable = new dataType;
```

---

delete Operator

Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.

For this, the `delete` operator is used. It returns the memory to the operating system. This is known as **memory deallocation**.

The syntax for this operator is

```
delete pointerVariable;
```

Consider the code:

```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
```

```
// for an int variable
pointVar = new int;

// assign value to the variable memory
*pointVar = 45;

// print the value stored in memory
cout << *pointVar; // Output: 45

// deallocate the memory
delete pointVar;
```

Here, we have dynamically allocated memory for an `int` variable using the pointer `pointVar`.
After printing the contents of `pointVar`, we deallocated the memory using `delete`.

> **Note**: If the program uses a large amount of unwanted memory using `new`, the system may crash because there will be no memory available for the operating system. In this case, the `delete` operator can help the system from crash.

Example 1: C++ Dynamic Memory Allocation

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

int main() {

  // declare an int pointer
  int* pointInt;

  // declare a float pointer
  float* pointFloat;

  // dynamically allocate memory
```

```
  pointInt = new int;
  pointFloat = new float;

  // assigning value to the memory
  *pointInt = 45;
  *pointFloat = 45.45f;

  cout << *pointInt << endl;
  cout << *pointFloat << endl;

  // deallocate the memory
  delete pointInt;
  delete pointFloat;

  return 0;
}
```

Run Code

Output

```
45
45.45
```

In this program, we dynamically allocated memory to two variables of `int` and `float` types. After assigning values to them and printing them, we finally deallocate the memories using the code

```
delete pointInt;
delete pointFloat;
```

Note: Dynamic memory allocation can make memory management more efficient.
Especially for arrays, where a lot of the times we don't know the size of the array until the run time.

Example 2: C++ new and delete Operator for Arrays

```
// C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user
```

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

int main() {

  int num;
  cout << "Enter total number of students: ";
  cin >> num;
  float* ptr;

  // memory allocation of num number of floats
  ptr = new float[num];

  cout << "Enter GPA of students." << endl;
  for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ": ";
    cin >> *(ptr + i);
  }

  cout << "\nDisplaying GPA of students." << endl;
  for (int i = 0; i < num; ++i) {
    cout << "Student" << i + 1 << ": " << *(ptr + i) << endl;
  }

  // ptr memory is released
  delete[] ptr;

  return 0;
}
```

Run Code

Output

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9

Displaying GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
```

```
Student4: 2.9
```

In this program, we have asked the user to enter the number of students and store it in the `num` variable.
Then, we have allocated the memory dynamically for the `float` array using `new`.
We enter data into the array (and later print them) using pointer notation.

After we no longer need the array, we deallocate the array memory using the code `delete[] ptr;`.
Notice the use of `[]` after `delete`. We use the square brackets `[]` in order to denote that the memory deallocation is that of an array.

Example 3: C++ new and delete Operator for Objects

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

class Student {
  private:
    int age;

  public:

    // constructor initializes age to 12
    Student() : age(12) {}

    void getAge() {
      cout << "Age = " << age << endl;
    }
};

int main() {

  // dynamically declare Student object
  Student* ptr = new Student();
```

```
  // call getAge() function
  ptr->getAge();

  // ptr memory is released
  delete ptr;

  return 0;
}
```

Run Code

Output

```
Age = 12
```

In this program, we have created a `Student` class that has a private variable `age`.

We have initialized `age` to `12` in the default constructor `Student()` and print its value with the function `getAge()`.

In `main()`, we have created a `Student` object using the `new` operator and use the pointer `ptr` to point to its address.

The moment the object is created, the `Student()` constructor initializes `age` to `12`.

We then call the `getAge()` function using the code:

```
ptr->getAge();
```

Notice the arrow operator `->`. This operator is used to access class members using pointers.