Data Types, Arrays, Pointers

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.

> There are a number of built-in types in C++.
> Tap **Continue** to learn more!

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
```

> You can implement some logic for illegal expressions by overloading operators. You'll learn about it later.

Numeric Data Types

Numeric data types include:
Integers (whole numbers), such as -7, 42.
Floating point numbers, such as 3.14, -42.67.

> We'll explain more about data types in the lessons to come.

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.

> In C++, single quotation marks indicate a **character**; double quotes create a **string** literal.
> While 'a' is a single a character literal, "a" is a string literal.

Booleans

The Boolean data type returns just two possible values: **true** (1) and **false** (0).

> **Conditional expressions** are an example of Boolean data type.

Integers

The integer type holds non-fractional numbers, which can be positive or negative. Examples of integers would include 42, -42, and similar numbers.

> The size of the integer type varies according to the architecture of the system on which the program runs, although 4 bytes is the minimum size in most modern system architectures.

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.

For example:

```
unsigned long int a;
```

> The integer data type reserves 4-8 bytes depending on the operating system.

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.
For example:

```
double temp = 4.21;
```

> Floating point data types are always **signed**, which means that they have the capability to hold both positive and negative values.

Strings

A string is an ordered sequence of characters, enclosed in **double quotation marks**.
It is part of the Standard Library.
You need to include the **<string>** library to use the string data type. Alternatively, you can use a library that includes the string library.

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

int main() {
  string a = "I am learning C++";
  return 0;
}
```

Try It Yourself

The <string> library is included in the <iostream> library, so you don't need to include <string> separately, if you already use <iostream>.

Characters

A **char** variable holds a 1-byte integer. However, instead of interpreting the value of the **char** as an integer, the value of a char variable is typically interpreted as an ASCII character.

A character is enclosed between **single quotes** (such as 'a', 'b', etc).
For example:

```
char test = 'S';
```

American Standard Code for Information Interchange (ASCII) is a character-encoding scheme that is used to represent text in computers.

Booleans

Boolean variables only have two possible values: **true** (1) and **false** (0).
To declare a boolean variable, we use the keyword **bool**.

```
bool online = false;
bool logged_in = true;
```

If a Boolean value is assigned to an integer, **true** becomes 1 and **false** becomes 0.
If an integer value is assigned to a Boolean, 0 becomes **false** and any value that has a non-zero value becomes **true**.

Variable Naming Rules

Use the following rules when naming variables:
- All variable names must begin with a letter of the alphabet or an underscore(_).
- After the initial letter, variable names can contain additional letters, as well as numbers. Blank spaces or special characters are not allowed in variable names.

There are two known naming conventions:
Pascal case: The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. For example: **BackColor**
Camel case: The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: **backColor**

Case-Sensitivity

C++ is **case-sensitive**, which means that an identifier written in uppercase is not equivalent to another one with the same name in lowercase.
For example, *myvariable* is not the same as *MYVARIABLE* and not the same as *MyVariable*.
These are three **different** variables.

> Choose variable names that suggest the usage, for example: firstName, lastName.

Variable Naming Rules

C++ keyword (reserved word) cannot be used as variable names.
For example, int, float, **double**, cout cannot be used as a variable name.

> There is no real limit on the length of the variable name (depends on the environment), but try to keep your variable names **practical** and **meaningful**.

Arrays

An array is used to store a collection of data, but it may be useful to think of an array as a collection of variables that are all of the **same type**.
Instead of declaring multiple variables and storing individual values, you can declare a single array to store all the values.
When declaring an array, specify its element types, as well as the number of elements it will hold.
For example:

```
int a[5];
```

In the example above, variable **a** was declared as an array of five integer values [specified in square brackets].
You can initialize the array by specifying the values it holds:

```
int b[5] = {11, 45, 62, 70, 88};
```

The values are provided in a **comma** separated list, enclosed in **{curly braces}**.

> The number of values between braces { } must not exceed the number of the elements declared within the square brackets [ ].

Initializing Arrays

If you omit the size of the array, an array just big enough to hold the initialization is created.
For example:

```
int b[] = {11, 45, 62, 70, 88};
```

This creates an identical array to the one created in the previous example.

Each element, or member, of the array has an **index**, which pinpoints the element's specific position.
The array's first member has the index of 0, the second has the index of 1.
So, for the array **b** that we declared above:

11	45	62	70	88
[0]	[1]	[2]	[3]	[4]

To access array elements, index the array name by placing the element's index in square brackets following the array name.
For example:

```
int b[] = {11, 45, 62, 70, 88};

cout << b[0] << endl;
// Outputs 11

cout<< b[3] << endl;
// Outputs 70
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Accessing Array Elements

Index numbers may also be used to assign a new value to an element.

```
int b[] = {11, 45, 62, 70, 88};
b[2] = 42;
```

This assigns a value of 42 to the array's third element.

Always remember that the list of elements always begins with the index of 0.

Arrays in Loops

It's occasionally necessary to iterate over the elements of an array, assigning the elements values based on certain calculations.

Usually, this is accomplished using a **loop**.

Arrays in Loops

Let's declare an array, that is going to store 5 integers, and assign a value to each element using the **for** loop:

```
int myArr[5];

for(int x=0; x<5; x++) {
  myArr[x] = 42;
}
```

Each element in the array is assigned the value of 42.
The **x** variable in the for loop is used as the index for the array.

> The last index in the array is 4, so the **for** loop condition is x<5.

Arrays in Loops

Let's output each index and corresponding value in the array.

```
int myArr[5];

for(int x=0; x<5; x++) {
  myArr[x] = 42;

  cout << x << ": " << myArr[x] << endl;
}

/* Outputs
0: 42
1: 42
2: 42
3: 42
4: 42
*/
```

Try It Yourself

> Tap **Try It Yourself** to play around with the code!

Arrays in Calculations

The following code creates a program that uses a **for** loop to calculate the sum of all elements of an array.

```
int arr[] = {11, 35, 62, 555, 989};
int sum = 0;

for (int x = 0; x < 5; x++) {
  sum += arr[x];
}

cout << sum << endl;
//Outputs 1652
```

Try It Yourself

To review, we declared an array and a variable **sum** that will hold the sum of the elements. Next, we utilized a **for** loop to iterate through each element of the array, and added the corresponding element's value to our **sum** variable.

> In the array, the first element's index is 0, so the **for** loop initializes the **x** variable to 0.

Multi-Dimensional Arrays

A **multi-dimensional** array holds one or more arrays. Declare a multidimensional array as follows.

```
type name[size1][size2]...[sizeN];
```

Here, we've created a **two-dimensional** 3x4 integer array:

```
int x[3][4];
```

Visualize this array as a table composed of 3 rows, and 4 columns.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Remember element counting always starts from 0.

Two-Dimensional Arrays

Multi-dimensional arrays may be initialized by specifying bracketed values for each row.
Following is an array with 2 rows and 3 columns:

```
int x[2][3] = {
  {2, 3, 4}, // 1st row
  {8, 9, 10} // 2nd row
};
```

You can also write the same initialization using just one row.

```
int x[2][3] = {{2, 3, 4}, {8, 9, 10}};
```

The elements are accessed by using the row index and column index of the array.
For example:

```
int x[2][3] = {{2, 3, 4}, {8, 9, 10}};
cout << x[0][2] << endl;

//Outputs 4
```

Try It Yourself

The first index 0 refers to the first row. The second index 2 refers to the 3rd element of the first row, which is 4.

Multi-Dimensional Arrays

Arrays can contain an unlimited number of dimensions.

```
string threeD[42][8][3];
```

The example above declares a **three-dimensional** array of strings. As we did previously, it's possible to use index numbers to access and modify the elements.

Arrays more than three dimensions are harder to manage.

Pointers

Every variable is a **memory** location, which has its **address** defined.
That address can be accessed using the **ampersand (&)** operator (also called the address-of operator), which denotes an **address in memory**.

For example:

```
int score = 5;
cout << &score << endl;

//Outputs "0x29fee8"
```

Try It Yourself

This outputs the **memory address**, which stores the variable **score**.

Pointers

A pointer is a variable, with the address of another variable as its value.
In C++, pointers help make certain tasks easier to perform. Other tasks, such as dynamic memory allocation, cannot be performed without using pointers.

All pointers share the same data type - a long **hexadecimal** number that represents a memory address.

The only difference between pointers of different data types is the data type of the variable that the pointer points to.

Pointers

A **pointer** is a variable, and like any other variable, it must be declared before you can work with it. The **asterisk** sign is used to declare a pointer (the same asterisk that you use for multiplication), however, in this statement the asterisk is being used to designate a variable as a pointer. Following are valid pointer declarations:

```
int *ip;  // pointer to an integer
double *dp;   // pointer to a double
float *fp;  // pointer to a float
char *ch;  // pointer to a character
```

Just like with variables, we give the pointers a name and define the type, to which the pointer points to.

> The asterisk sign can be placed next to the data type, or the variable name, or in the middle.

Using Pointers

Here, we assign the address of a variable to the pointer.

```
int score = 5;
int *scorePtr;
scorePtr = &score;

cout << scorePtr << endl;

//Outputs "0x29fee8"
```

Try It Yourself

The code above declares a pointer to an integer called **scorePtr**, and assigns to it the memory location of the **score** variable using the ampersand (address-of) operator.

> Now, **scorePtr's** value is the memory location of **score**.

Pointer Operations

There are two operators for pointers:
Address-of operator (&): returns the memory address of its operand.
Contents-of (or **dereference**) operator (*): returns the value of the variable located at the address specified by its operand.

For example:

```
int var = 50;
int  *p;
p = &var;
```

```
cout << var << endl;
// Outputs 50 (the value of var)

cout << p << endl;
// Outputs 0x29fee8 (var's memory location)

cout << *p << endl;
/* Outputs 50 (the value of the variable
 stored in the pointer p) */
```

Try It Yourself

The asterisk (*) is used in declaring a pointer for the simple purpose of indicating that it is a pointer (The asterisk is part of its type compound specifier). Don't confuse this with the **dereference** operator, which is used to obtain the value located at the specified address. They are simply two different things represented with the same sign.

Dereferencing

The **dereference** operator (*) is basically an **alias** for the variable the pointer points to.
For example:

```
int x = 5;
int *p = &x;

x = x + 4;
x = *p + 4;
*p = *p + 4;
```

All three of the preceding statements are equivalent, and return the same result. We can access the variable by dereferencing the variable's pointer.

As **p** is pointing to the variable **x**, dereferencing the pointer (***p**) is representing exactly the same as the variable **x**.

Static & Dynamic Memory

To be successful as a C++ programmer, it's essential to have a good understanding of how **dynamic memory** works.
In a C++ program, memory is divided into two parts:
The stack: All of your local variables take up memory from the stack.
The heap: Unused program memory that can be used when the program runs to **dynamically** allocate the memory.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time. You can allocate memory at run time within the heap for the variable of a given type using the **new** operator, which returns the address of the space allocated.

```
new int;
```

This allocates the memory size necessary for storing an **integer** on the **heap**, and returns that address.

Dynamic Memory

The allocated address can be stored in a pointer, which can then be dereferenced to access the variable.
Example:

```
int *p = new int;
*p = 5;
```

We have dynamically allocated memory for an integer, and assigned it a value of 5.

> The pointer **p** is stored in the **stack** as a local variable, and holds the **heap**'s allocated address as its value. The value of 5 is stored at that address in the heap.

Dynamic Memory

For local variables on the **stack**, managing memory is carried out automatically.
On the heap, it's necessary to manually handle the dynamically allocated memory, and use the **delete** operator to free up the memory when it's no longer needed.

```
delete pointer;
```

This statement releases the memory pointed to by pointer.

For example:

```
int *p = new int; // request memory
*p = 5; // store value

cout << *p << endl; // use value

delete p; // free up the memory
```

Try It Yourself

> Forgetting to free up memory that has been allocated with the **new** keyword will result in memory leaks, because that memory will stay allocated until the program shuts down.

Dangling Pointers

The **delete** operator frees up the memory allocated for the variable, but does not delete the pointer itself, as the pointer is stored on the stack.

Pointers that are left pointing to non-existent memory locations are called **dangling pointers**.
For example:

```
int *p = new int; // request memory
*p = 5; // store value

delete p; // free up the memory
// now p is a dangling pointer

p = new int; // reuse for a new address
```

Try It Yourself

The **NULL** pointer is a constant with a value of zero that is defined in several of the standard libraries, including iostream.
It's a good practice to assign NULL to a pointer variable when you declare it, in case you do not have exact address to be assigned. A pointer assigned NULL is called a **null pointer**. For example: int *ptr = NULL;

Dynamic Memory

Dynamic memory can also be allocated for arrays.
For example:

```
int *p = NULL; // Pointer initialized with null
p = new int[20]; // Request memory
delete [] p; // Delete array pointed to by p
```

Try It Yourself

Note the **brackets** in the syntax.

Dynamic memory allocation is useful in many situations, such as when your program depends on input. As an example, when your program needs to read an image file, it doesn't know in advance the size of the image file and the memory necessary to store the image.

---

sizeof

While the size allocated for varying data types depends on the architecture of the computer you use to run your programs, C++ does guarantee a minimum size for the basic data types:

Category	Type	Minimum Size
boolean	bool	1 byte
character	char	1 byte
integer	short	2 bytes
	int	2 bytes
	long	4 bytes
	long long	8 bytes
floating point	float	4 bytes
	double	8 bytes
	long double	8 bytes

The **sizeof** operator can be used to get a variable or data type's size, in bytes.
Syntax:

```
sizeof (data type)
```

The **sizeof** operator determines and returns the size of either a type or a variable in bytes.
For example:

```
cout << "char: " << sizeof(char) << endl;
cout << "int: " << sizeof(int) << endl;
cout << "float: " << sizeof(float) << endl;
cout << "double: " << sizeof(double) << endl;
int var = 50;
cout << "var: " << sizeof(var) << endl;

/* Outputs
char: 1
int: 4
float: 4
double: 8
var: 4
*/
```

Try It Yourself

Output values may vary, according to the computer and compiler used.

Size of an Array

The C++ **sizeof** operator is also used to determine the size of an **array**.
For example:

```
double myArr[10];
cout << sizeof(myArr) << endl;

//Outputs 80
```

Try It Yourself

On our machine, **double** takes 8 bytes. The array stores 10 doubles, so the entire array occupies 80 (8*10) bytes in the memory.

In addition, divide the total number of bytes in the array by the number of bytes in a single element to learn how many elements you have in the array.
For example:

```
int numbers[100];
cout << sizeof(numbers) / sizeof(numbers[0]);

// Outputs 100
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

End.