5. DATA TYPES

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.
You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	Wchar_t

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,647 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.
Following is the example, which will produce correct size of various data types on your computer.

```
#include <iostream>
using namespace std;
int main()
{
cout << "Size of char : " << sizeof(char) << endl;
cout << "Size of int : " << sizeof(int) << endl;
cout << "Size of short int : " << sizeof(short int) << endl;
cout << "Size of long int : " << sizeof(long int) << endl;
cout << "Size of float : " << sizeof(float) << endl;
cout << "Size of double : " << sizeof(double) << endl;
cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
return 0;
}
```

This example uses `endl`, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using `sizeof()` function to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine:

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

typedef Declarations

You can create a new name for an existing type using `typedef`. Following is the simple syntax to define a new type using typedef:

C++

```
typedef type newname;
```
For example, the following tells the compiler that feet is another name for int:
```
typedef int feet;
```
Now, the following declaration is perfectly legal and creates an integer variable called distance:
```
feet distance;
```
Enumerated Types
An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.
Creating an enumeration requires the use of the keyword `enum`. The general form of an enumeration type is:
```
enum enum-name { list of names } var-list;
```
Here, the enum-name is the enumeration's type name. The list of names is comma separated.
For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".
```
enum color { red, green, blue } c;
c = blue;
```
By default, the value of the first name is 0, the second name has the value 1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, `green` will have the value 5.
```
enum color { red, green=5, blue };
```
Here, `blue` will have a value of 6 because each name will be one greater than the one that precedes it.

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive:

Type	Description
boolean	Stores either value true or false.
char	Typically a single octet (one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.
wchar_t	A wide character type.

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Reference, Data structures, and Classes. Following section will cover how to define, declare and use various types of variables.

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, type must be a valid C++ data type including char, w_char, int, float, double, bool or any user-defined object, etc., and variable_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

The line int i, j, k; both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.
Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:

```
extern int d = 3, f = 5; // declaration of d and f.
int d = 3, f = 5; // definition and initializing d and f.
byte z = 22; // definition and initializes z.
char x = 'x'; // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

Variable Declaration in C++

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program.
A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of c++.

the program. You will use `extern` keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Example:

Try the following example where a variable has been declared at the top, but it has been defined inside the main function:

```
#include <iostream>
using namespace std;
// Variable declaration:
extern int a, b;
extern int c;
extern float f;
int main ()
{
// Variable definition:
int a, b;
int c;
float f;
// actual initialization
a = 10;
b = 20;
c = a + b;
cout << c << endl ;
f = 70.0/3.0;
cout << f << endl ;
return 0;
} C++
```

When the above code is compiled and executed, it produces the following result:

```
30
23.3333
```

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example:

```
// function declaration
int func();
int main()
{
// function call
int i = func();
}
// function definition
int func()
{
return 0;
}
```

Lvalues and Rvalues

There are two kinds of expressions in C++:

- lvalue : Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- rvalue : The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and cannot appear on the left-hand side. Following is a valid statement:

```
int g = 20; C++
```

But the following is not a valid statement and would generate compile-time error:

```
10 = 20; C++
```

A scope is a region of the program and broadly speaking there are three places, where variables can be declared:
☐ Inside a function or a block which is called local variables,
☐ In the definition of function parameters which is called formal parameters.
☐ Outside of all functions which is called global variables.

We will learn what a function is, and it's parameter in subsequent chapters. Here let us explain what local and global variables are.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:

```
#include <iostream>
using namespace std;
int main ()
{
// Local variable declaration:
int a, b;
int c;
// actual initialization
a = 10;
b = 20;
c = a + b;
cout << c;
return 0;
}
```

7. VARIABLE SCOPE C++

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
#include <iostream>
using namespace std;
// Global variable declaration:
int g;
int main ()
{
// Local variable declaration:
int a, b;
// actual initialization
a = 10;
b = 20;
g = a + b;
cout << g;
return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example:

```
#include <iostream>
using namespace std;
// Global variable declaration:
int g = 20; C++
```

```
int main ()
{
// Local variable declaration:
int g = 10;
cout << g;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
10
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself.

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result. C++

Constants refer to fixed values that the program may not alter and they are called literals.
Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values. Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows:

Data Type	Initializer
int	0
char	'\0'
float	0
double	0
pointer	NULL

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.
An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.
Here are some examples of integer literals:

```
212 // Legal
215u // Legal
0xFeeL // Legal
078 // Illegal: 8 is not an octal digit
032UU // Illegal: cannot repeat a suffix
```

Following are other examples of various types of Integer literals:

```
85 // decimal
0213 // octal
0x4b // hexadecimal
30 // int
30u // unsigned int
30l // long
30ul // unsigned long
```

8. CONSTANTS/LITERALS C++

Constants refer to fixed values that the program may not alter and they are called literals.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values. Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212 // Legal
215u // Legal
0xFeeL // Legal
078 // Illegal: 8 is not an octal digit
032UU // Illegal: cannot repeat a suffix
```

Following are other examples of various types of Integer literals:

```
85 // decimal
0213 // octal
0x4b // hexadecimal
30 // int
30u // unsigned int
30l // long
30ul // unsigned long
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
3.14159 // Legal
314159E-5L // Legal
510E // Illegal: incomplete exponent
210f // Illegal: no decimal or exponent
.e55 // Illegal: missing integer or fraction
```

Boolean Literals

There are two Boolean literals and they are part of standard C++ keywords:

☐ A value of `true` representing true.

☐ A value of `false` representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

Character Literals

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in `wchar_t` type of variable. Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of `char` type.
A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

C++

\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show a few escape sequence characters:

```
#include <iostream>
using namespace std;
int main()
{
cout << "Hello\tWorld\n\n";
return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result:

```
C++
```

```
Hello World
```

String Literals

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
```

Defining Constants

There are two simple ways in C++ to define constants:

☐ Using `#define` preprocessor.

☐ Using `const` keyword.

The #define Preprocessor

Following is the form to use #define preprocessor to define a constant:

```
#define identifier value
```

Following example explains it in detail:

```
#include <iostream>
using namespace std;
#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n' C++
30
```

```
int main()
{
int area;
area = LENGTH * WIDTH;
cout << area;
cout << NEWLINE;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
50
```

The const Keyword

You can use `const` prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

Following example explains it in detail:

```
#include <iostream>
using namespace std;
int main()
{
const int LENGTH = 10;
const int WIDTH = 5;
const char NEWLINE = '\n';
int area;
area = LENGTH * WIDTH;
cout << area;
cout << NEWLINE;
return 0;
```

```
}
```
When the above code is compiled and executed, it produces the following result:
```
50
```
Note that it is a good programming practice to define constants in CAPITALS.