**Data Types:**

Any programming language has two types of data, elementary (primitive) data type and structured data type. We will take C++ language as an example to illustrate these types.

1. **Elementary data types.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Bytes</th>
<th>Description</th>
<th>Range*</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>character or integer 8 bits length.</td>
<td>signed: -126 to 127</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned: 0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>integer 16 bits length.</td>
<td>signed: -32768 to 32767</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned: 0 to 65535</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>integer 32 bits length.</td>
<td>signed: 2147483648 to 2147483647</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned: 0 to 4294967295</td>
</tr>
<tr>
<td>int</td>
<td>-</td>
<td>Integer. Its length traditionally depends on the length of the system's word type, thus in MSDOS it is 16 bits long, whereas in 32 bit systems (like Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes).</td>
<td>See short, long</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>floating point number.</td>
<td>3.4e + 8/-38 (7 digits)</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>double precision floating point number.</td>
<td>1.7e + 8/-308 (15 digits)</td>
</tr>
<tr>
<td>long double</td>
<td>10</td>
<td>long double precision floating point number.</td>
<td>1.2e + 8/-4532 (19 digits)</td>
</tr>
<tr>
<td>bool</td>
<td>1</td>
<td>Boolean value. It can take one of two values: true or false. NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it.</td>
<td>true or false</td>
</tr>
<tr>
<td>wchar_t</td>
<td>2</td>
<td>Wide character. It is designed as a type to store international characters of a two-byte character set. NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it.</td>
<td>wide characters</td>
</tr>
</tbody>
</table>

2. **Structured data type**

2.1 Arrays

In general, to read or write an array in C++, it can be done as follows:

For reading an array called `a` of `n` elements

```cpp
for (i = 0; i < n; i++)
    cin >>"%d", &a[i];
```

For writing an array called `a` of `n` elements

```cpp
for (i = 0; i < n; i++)
    cout <<"%d", &a[i];
```

where `i` is the counter number of the loop.

2.1.1 One dimensional arrays

C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as `number0, number1, ...`, and `number99`, you declare one array variable such as `numbers[0], numbers[1], ... numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

**Declaring Arrays:**

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

```cpp
type arrayName [ arraySize ];
```
This is called a single-dimension array. The \texttt{arraySize} must be an integer constant greater than zero and \texttt{type} can be any valid C++ data type. For example, to declare a 10-element array called \texttt{balance} of type double, use this statement:
\begin{verbatim}
double balance[10];
\end{verbatim}

**Initializing Arrays:**

You can initialize C++ array elements either one by one or using a single statement as follows:
\begin{verbatim}
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
\end{verbatim}

The number of values between braces \{\} can not be larger than the number of elements that we declare for the array between square brackets []. Following is an example to assign a single element of the array:

\begin{verbatim}
if you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:
\end{verbatim}
\begin{verbatim}
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
\end{verbatim}

You will create exactly the same array as you did in the previous example.

\begin{verbatim}
balance[4] = 50.0;
\end{verbatim}

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

\begin{center}
\begin{tabular}{c|c|c|c|c}
\hline
0 & 1 & 2 & 3 & 4 \\
\hline
1000.0 & 2.0 & 3.4 & 7.0 & 50.0 \\
\hline
\end{tabular}
\end{center}

**Accessing Array Elements:**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:
\begin{verbatim}
double salary = balance[9];
\end{verbatim}

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

\begin{verbatim}
#include <iostream>
using namespace std;

#include <iomanip>
using std::setw;

int main ()
{
    int n[ 10 ]; // n is an array of 10 integers
    // initialize elements of array n to 0
    for ( int i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; // set element at location i to i + 100
    }
    cout << "Element" << setw( 13 ) << "Value" << endl;
    // output each array element's value
    for ( int j = 0; j < 10; j++ )
    {
        cout << setw( 7 )<" j " << setw( 13 ) << n[ j ] << endl;
    }
    return 0;
}
\end{verbatim}

Using \texttt{setw()} in C++ is a good way to make text output look sharper. The \texttt{iomanip} library provides the user with the \texttt{setw()} function which creates sized fields.

This program makes use of \texttt{setw()} function to format the output. When the above code is compiled and executed, it produces the following result:

\begin{verbatim}
Element Value
0 100
1 101
2 102
3 103
4 104
5 105
6 106
\end{verbatim}
2.1.2 Multi-dimensional array

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

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

For example, the following declaration creates a three-dimensional 5 x 10 x 4 integer array:

```
int threedim[5][10][4];
```

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x.y, you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where `type` can be any valid C++ data type and `arrayName` will be a valid C++ identifier. A two-dimensional array can be thought of as a table, which will have x number of rows and y number of columns. A 2-dimensional array `a`, which contains three rows and four columns can be shown as below:

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
</tr>
<tr>
<td>Row 1</td>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
</tr>
</tbody>
</table>

Thus, every element in array `a` is identified by an element name of the form `a[i][j]`, where `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify each element in `a`.

Initializing Two-Dimensional Arrays:

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3}, /* initializers for row indexed by 0 */  
    {4, 5, 6, 7}, /* initializers for row indexed by 1 */  
    {8, 9, 10, 11} /* initializers for row indexed by 2 */  
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[4][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram.
return 0;

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

```
[a][0][0]: 0
[a][0][1]: 0
[a][1][0]: 1
[a][1][1]: 2
[a][2][0]: 2
[a][2][1]: 4
[a][3][0]: 3
[a][3][1]: 6
[a][4][0]: 4
[a][4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

### 2.2 Vectors

In C++, vectors are one type of dynamically-allocated container from the standard template library (STL). They are capable of storing multiple elements of any defined datatype, and do so in a contiguous block of memory. Unlike raw arrays, vectors are access-safe as long as you use the built-in access methods associated with them. Thus, overrunning the end of a vector will throw an exception rather than crashing the program. For quickly writing access-safe code that requires very little in the way of programmer intervention for memory management, they are hard to beat. There are reasons to use other containers from the standard template library (such as lists or double-ended queues), but vectors are generally the first choice of most programmers for their versatility and speed.

Vectors are declared with the following syntax:

```
vector<type> variable_name (number_of_elements);
```

The number of elements is optional. You could declare it like this:

```
vector<type> variable_name;
```

And that would declare an empty vector — a vector that contains zero elements.

The argument `type` in angle-brackets indicates the data type of the elements of the vector; `variable_name` is the name that we assign to the vector, and the optional `number_of_elements` may be provided to indicate how many elements the vector will initially contain.

Below are several examples of `vector` declarations:

```
vector<int> values (5);      // Declares a vector of 5 integers
vector<double> grades (20);  // Declares a vector of 20 doubles
vector<string> names;        // Declares a vector of strings,
                              // initially empty (contains 0 strings)
```

When using vectors in our programs, we must provide the appropriate `#include` directive at the top, since vectors are a Standard Library facility, and not a built-in part of the core language:

```
#include <vector>
```

After a vector has been declared specifying a certain number of elements, we can refer to individual elements in the vector using square brackets to provide a subscript or index, as shown below:

```
grades[5]
```
When using a vector or array followed by square brackets with a subscript, the resulting expression refers to one individual element of the vector or array, as opposed to the group of values, and roughly speaking, you can use that expression as you would use a variable of the corresponding data type. In the above example, the data type of the expression `grades[5]` is `double`, so you can use it as you would use a variable of type double — you can assign a value to it (a numeric value, with or without decimals), or you can retrieve the value, use it for arithmetic operations, etc.

The above extends to other data types as well; if we have a `vector` of `strings` called `names`, the expression `names[0]` is a `string`, referring to the first element in the vector `names`. We can do anything with this expression that we would do with a `string` variable. For instance, the expression `names[0].length()` gives us the length of this string.

An important condition for the index or subscript is that it must indicate a valid element in the vector. Elements in a vector are “numbered” starting with element 0. This means that valid subscript values are numbers between 0 and `size-1`, where `size` is the number of elements of the vector. For the example above of `grades`, valid subscripts are between 0 and 19.

The following fragment shows an example of a program that asks the user for marks for a group of 20 students and stores them in a vector.

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

int main()
{
    vector<double> student_marks(20);

    for (vector<double>::size_type i = 0; i < 20; i++)
    {
        cout << "Enter marks for student #" << i+1
             << ": " << flush;
        cin >> student_marks[i];
    }
    // ... Do some stuff with the values
    return 0;
}
```

The first statement declares a vector called `student_marks` with capacity to hold 20 values of type `double`. These values can be accessed individually as `student_marks[0]` to `student_marks[19]`. The `for` loop has the counter `i` go from 0 to 19, allowing access to each individual element in a sequential manner, starting at 0 and going through each value from 0 to 19, inclusively.

Notice the data type for the subscript, `vector<double>::size_type`. As with strings, class `vector<type>` provides `size_type` to represent positions and sizes. It is always recommended that you use this data type when dealing with vectors.

`for` loops usually go hand in hand with the use of vectors or arrays, as they provide a convenient way to access every element, one at a time, using the loop control variable as the subscript. This does not mean that we must use `for` loops whenever we require access to the elements of a vector — it only means that quite often, a `for` loop provides a convenient approach and we choose it as the mechanism to access the elements.

### 2.3 Pointers

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.
As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined:

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

int main ()
{
    int var1;
    char var2[10];

    cout << "Address of var1 variable: ";
    cout << &var1 << endl;

    cout << "Address of var2 variable: ";
    cout << &var2 << endl;

    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

| Address of var1 variable: 0xbfebd5c0 |
| Address of var2 variable: 0xbfebd5b6 |

**What Are Pointers?**
A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, `type` is the pointer's base type; it must be a valid C++ type and `var-name` is the name of the pointer variable. The asterisk you used to declare a pointer is 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 the valid pointer declaration:

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

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, 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 or constant that the pointer points to.

**Using Pointers in C++:**
There are few important operations, which we will do with the pointers very frequently. (a) we define a pointer variables (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

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

int main ()
{
```
int var = 20; // actual variable declaration.
int *ip;      // pointer variable

ip = &var;    // store address of var in pointer variable

cout << "Value of var variable: ";
cout << var << endl;

// print the address stored in ip pointer variable
cout << "Address stored in ip variable: ";
cout << ip << endl;

// access the value at the address available in pointer
cout << "Value of *ip variable: ";
cout << *ip << endl;

return 0;
}

When the above code is compiled and executed, it produces result something as follows:

<table>
<thead>
<tr>
<th>Value of var variable: 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address stored in ip variable: 0xbfc601ac</td>
</tr>
<tr>
<td>Value of *ip variable: 20</td>
</tr>
</tbody>
</table>