C++_Lecture Note 1

Fundamental Data Types of C++ Program

Fundamental Data Types

The values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones. Our program does not need to know the exact location where a variable is stored; it can simply refer to it by its name. What the program needs to be aware of is the kind of data stored in the variable. It's not the same to store a simple integer as it is to store a letter or a large floating-point number; even though they are all represented using zeros and ones, they are not interpreted in the same way, and in many cases, they don't occupy the same amount of memory.

Fundamental data types are basic types implemented directly by the language that represent the basic storage units supported natively by most systems. They can mainly be classified into:

- **Character types:** They can represent a single character, such as `'A'` or `'$'`. The most basic type is `char`, which is a one-byte character. Other types are also provided for wider characters.
- **Numerical integer types:** They can store a whole number value, such as `7` or `1024`. They exist in a variety of sizes, and can either be *signed* or *unsigned*, depending on whether they support negative values or not.
- **Floating-point types:** They can represent real values, such as `3.14` or `0.01`, with different levels of precision, depending on which of the three floating-point types is used.
- **Boolean type:** The boolean type, known in C++ as `bool`, can only represent one of two states, `true` or `false`.

Here is the complete list of fundamental types in C++:

Group	Type names*	Notes on size / precision
Character types	**`char`**	Exactly one byte in size. At least 8 bits.
	`char16_t`	Not smaller than `char`. At least 16 bits.
	`char32_t`	Not smaller than `char16_t`. At least 32 bits.
	`wchar_t`	Can represent the largest supported character set.
Integer types (signed)	**`signed char`**	Same size as `char`. At least 8 bits.
	`signed` **`short`** *`int`*	Not smaller than `char`. At least 16 bits.
	`signed` **`int`**	Not smaller than `short`. At least 16 bits.
	`signed` **`long`** *`int`*	Not smaller than `int`. At least 32 bits.
	`signed` **`long long`** *`int`*	Not smaller than `long`. At least 64 bits.
Integer types (unsigned)	**`unsigned char`**	(same size as their signed counterparts)
	`unsigned short` *`int`*	

	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Floating-point types	**float**	
	double	Precision not less than `float`
	long double	Precision not less than `double`
Boolean type	**bool**	
Void type	**void**	no storage
Null pointer	**decltype(nullptr)**	

* The names of certain integer types can be abbreviated without their signed and int components - only the part not in italics is required to identify the type, the part in italics is optional.
i..e., *signed* short *int* can be abbreviated as signed short, short int, or simply short; they all identify the same fundamental type.

Within each of the groups above, the difference between types is only their size (i.e., how much they occupy in memory): the first type in each group is the smallest, and the last is the largest, with each type being at least as large as the one preceding it in the same group. Other than that, the types in a group have the same properties.

Note in the panel above that other than char (which has a size of exactly one byte), none of the fundamental types has a standard size specified (but a minimum size, at most). Therefore, the type is not required (and in many cases is not) exactly this minimum size. This does not mean that these types are of an undetermined size, but that there is no standard size across all compilers and machines; each compiler implementation may specify the sizes for these types that fit the best the architecture where the program is going to run. This rather generic size specification for types gives the C++ language a lot of flexibility to be adapted to work optimally in all kinds of platforms, both present and future.

Type sizes above are expressed in bits; the more bits a type has, the more distinct values it can represent, but at the same time, also consumes more space in memory:

Size	Unique representable values	Notes
8-bit	256	= 2^8
16-bit	65 536	= 2^{16}
32-bit	4 294 967 296	= 2^{32} (~4 billion)
64-bit	18 446 744 073 709 551 616	= 2^{64} (~18 billion billion)

For integer types, having more representable values means that the range of values they can represent is greater; for example, a 16-bit unsigned integer would be able to represent 65536 distinct values in the range 0 to 65535, while its signed counterpart would be able to represent, on most cases, values between -32768 and 32767. Note that the range of positive values is approximately halved in signed types compared to unsigned types, due to the fact that one of the 16 bits is used for the sign; this is a relatively modest difference in range, and seldom justifies the use of unsigned types based purely on the range of positive values they can represent.

For floating-point types, the size affects their precision, by having more or less bits for their significant and exponent.

If the size or precision of the type is not a concern, then char, int, and double are typically selected to represent characters, integers, and floating-point values, respectively. The other types in their respective groups are only used in very particular cases.

The properties of fundamental types in a particular system and compiler implementation can be obtained by using the numeric_limits classes (see standard header <limits>). If for some reason, types of specific sizes are needed, the library defines certain fixed-size type aliases in header <cstdint>.

The types described above (characters, integers, floating-point, and boolean) are collectively known as arithmetic types. But two additional fundamental types exist: void, which identifies the lack of type; and the type nullptr, which is a special type of pointer. Both types will be discussed

further in a coming chapter about pointers.

C++ supports a wide variety of types based on the fundamental types discussed above; these other types are known as *compound data types*, and are one of the main strengths of the C++ language. We will also see them in more detail in future chapters.

Declaration of Variable

C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use. This informs the compiler the size to reserve in memory for the variable and how to interpret its value. The syntax to declare a new variable in C++ is straightforward: we simply write the type followed by the variable name (i.e., its identifier).

For example:

```
1 int a;
2 float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and mynumber can be used within the rest of their scope in the program.
If declaring more than one variable of the same type, they can all be declared in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (`a`, `b` and `c`), all of them of type `int`, and has exactly the same meaning as:

```
1 int a;
2 int b;
3 int c;
```

To see what variable declarations look like in action within a program, let's have a look at the entire C++ code of the example about your mental memory proposed at the beginning of this chapter:

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

```
4
```

Don't be worried if something else than the variable declarations themselves look a bit strange to you. Most of it will be explained in more detail in coming chapters.

Initialization of Variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time. But it is possible for a variable to have a specific value from the moment it is declared. This is called the *initialization* of the variable.

In C++, there are three ways to initialize variables. They are all equivalent and are reminiscent of the evolution of the language over the years:

The first one, known as *c-like initialization* (because it is inherited from the C language), consists

of appending an equal sign followed by the value to which the variable is initialized:

type identifier = initial_value;
For example, to declare a variable of type int called x and initialize it to a value of zero from the same moment it is declared, we can write:

```
int x = 0;
```

A second method, known as *constructor initialization* (introduced by the C++ language), encloses the initial value between parentheses (()):

type identifier (initial_value);
For example:

```
int x (0);
```

Finally, a third method, known as *uniform initialization*, similar to the above, but using curly braces ({}) instead of parentheses (this was introduced by the revision of the C++ standard, in 2011):

type identifier {initial_value};
For example:

```
int x {0};
```

All three ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
```

```
6
```

```
  int a=5;                // initial value: 5
  int b(3);               // initial value: 3
  int c{2};               // initial value: 2
  int result;             // initial value
undetermined

  a = a + b;
  result = a - c;
  cout << result;

  return 0;
}
```

Type deduction auto and decitype

When a new variable is initialized, the compiler can figure out what the type of the variable is automatically by the initializer. For this, it suffices to use auto as the type specifier for the variable:

```
int foo = 0;
auto bar = foo;  // the same as: int bar = foo;
```

Here, bar is declared as having an auto type; therefore, the type of bar is the type of the value used to initialize it: in this case it uses the type of foo, which is int.

Variables that are not initialized can also make use of type deduction with the decltype specifier:

```
int foo = 0;
decltype(foo) bar;  // the same as: int bar;
```

Here, bar is declared as having the same type as foo.

auto and decltype are powerful features recently added to the language. But the type deduction features they introduce are meant to be used either when the type cannot be obtained by other means or when using it improves code readability. The two examples above were likely neither of these use cases. In fact they probably decreased readability, since, when reading the code, one

has to search for the type of foo to actually know the type of bar.

Introduction to Strings

Fundamental types represent the most basic types handled by the machines where the code may run. But one of the major strengths of the C++ language is its rich set of compound types, of which the fundamental types are mere building blocks.

An example of compound type is the string class. Variables of this type are able to store sequences of characters, such as words or sentences. A very useful feature!

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type, the program needs to include the header where the type is defined within the standard library (header <string>):

```
1  // my first string
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main ()
7  {
8    string mystring;
9    mystring = "This is a string";
10   cout << mystring;
11   return 0;
12 }
```

```
This is a string
```

As you can see in the previous example, strings can be initialized with any valid string literal, just like numerical type variables can be initialized to any valid numerical literal. As with fundamental types, all initialization formats are valid with strings:

```
1 string mystring = "This is a string";
2 string mystring ("This is a string");
3 string mystring {"This is a string"};
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and change its value during execution:

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string
content";
  cout << mystring << endl;
  mystring = "This is a different string
content";
  cout << mystring << endl;
  return 0;
}
```

```
This is the initial string content
This is a different string content
```

Note: inserting the endl manipulator **end**s the **l**ine (printing a newline character and flushing the stream).

The string class is a *compound type*. As you can see in the example above, *compound types* are used in the same way as *fundamental types*: the same syntax is used to declare variables and to initialize them.