C++_Lecture Note 2

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.

A **data type** (more commonly just called a **type**) tells the compiler what type of value (e.g. a number, a letter, text, etc…) the variable will store.
In the above example, our variable *x* was given type *int*, which means variable x will represent an integer value. An **integer** is a number that can be written without a fractional component, such as 4, 27, 0, -2, or -12. For short, we can say that *x* is an *integer variable*.
In C++, the type of a variable must be known at **compile-time** (when the program is compiled), and that type can not be changed without recompiling the program. This means an integer variable can only hold integer values. If you want to store some other kind of value, you'll need to use a different variable.
Integers are just one of many types that C++ supports out of the box. For illustrative purposes, here's another example of defining a variable using data type *double*:

```
1 | double width; // define a variable named width, of type double
```

C++ also allows you to create your own user-defined types. This is something we'll do a lot of in future lessons, and it's part of what makes C++ powerful.

Data types are the classifications for different kinds of data you can use in a program. Data types tell our variables what data they can store. Data types can also be classified into the following in C++:

- **Primitive data types:** these are the built-in data that you can use to declare variables. They include integer, character, boolean, floating point, double floating point, void, and wide character.

- **Derived data types:** these are derived from the primitive data types. They include function, reference, array, and pointer.
- **User-Defined data types:** these are defined by you, the programmer.

Variable

Variables are like containers that store values. To declare a variable, you must give it a value and a type using the correct keyword. All variables in C++ need a name, or identifier. There are some basic syntax rules to follow when making identifiers.

- Names are case sensitive
- Names can contain letters, numbers, and underscores
- Names must begin with a letter or an underscore
- Names cannot contain whitespaces or special characters (!, #, @, etc.)
- Names cannot use reserved keywords

There are six different types of variables:

```
int myNum = 5;               // Stores integers (whole numbers)
float myFloatNum = 5.99;     // Stores decimals loating point number
double myDoubleNum = 9.98;   // Floating point number
char myLetter = 'D';         // Stores single characters
bool myBoolean = true;       // Stores Boolean, values with a true o
r false state
string myText = "Hello";     // Stores strings of text
```

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:

```
int a;
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:

```
int a;
int b;
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;
```

```
4
```

```
20
21    // terminate the program:
22    return 0;
23  }
```

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 notes.

Variable assignment

After a variable has been defined, you can give it a value (in a separate statement) using the = *operator*. This process is called **copy assignment** (or just **assignment**) for short.

```
1  int width; // define an integer variable named width
2  width = 5; // copy assignment of value 5 into variable width
3
4  // variable width now has value 5
```

Copy assignment is named such because it copies the value on the right-hand side of the = *operator* to the variable on the left-hand side of the operator. The = *operator* is called the **assignment operator**.

Here’s an example where we use assignment twice:

```cpp
#include <iostream>

int main()
{
    int width;
    width = 5; // copy assignment of value 5 into variable width

    // variable width now has value 5

    width = 7; // change value stored in variable width to 7

    // variable width now has value 7

    return 0;
}
```

When we assign value 7 to variable *width*, the value 5 that was there previously is overwritten. Normal variables can only hold one value at a time.

Initialization

One downside of assignment is that it requires at least two statements: one to define the variable, and one to assign the value.

These two steps can be combined. When a variable is defined, you can also provide an initial value for the variable at the same time. This is called **initialization**. The value used to initialize a variable is called an **initializer**.
Initialization in C++ is surprisingly complex, so we'll present a simplified view here.

There are 4 basic ways to initialize variables in C++:

```cpp
int a; // no initializer
int b = 5; // initializer after equals sign
int c( 6 ); // initializer in parenthesis
int d { 7 }; // initializer in braces
```

You may see the above forms written with different spacing (e.g. `int d{7};`). Whether you use extra spaces for readability or not is a matter of personal preference.
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 is no initialization

The second 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 third 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 four ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
  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;
}
```

```
6
```

Default initialization

When no initialization value is provided (such as for variable *a* above), this is called **default initialization**. In most cases, default initialization leaves a variable with an indeterminate value.

Copy initialization

When an initializer is provided after an equals sign, this is called **copy initialization**. Copy initialization was inherited from the C language.

```
int width = 5; // copy initialization of value 5 into variable width
```

Much like copy assignment, this copies the value on the right-hand side of the equals to the variable being created on the left-hand side.

For simple types like int, copy initialization is efficient. However, when types get more complex, copy initialization can be inefficient.

Direct initialization

When an initializer is provided inside parenthesis, this is called **direct initialization**.

For simple data types (like int), copy and direct initialization are essentially the same. For more complicated types, direct initialization tends to be more efficient than copy initialization.

Brace initialization

Unfortunately, direct initialization can't be used for all types of initialization (such as initializing an object with a list of data). To provide a more consistent initialization mechanism, there's **brace initialization** (also called **uniform initialization** or **list initialization**) that uses curly braces.
Brace initialization comes in three forms:

```
1 | int width { 5 }; // direct brace initialization of value 5 into variable width (preferred)
2 | int height = { 6 }; // copy brace initialization of value 6 into variable height
3 | int depth {}; // value initialization (see next section)
```

Direct and copy brace initialization function almost identically, but the direct form is generally preferred.

Brace initialization has the added benefit of disallowing "narrowing" conversions. This means that if you try to use brace initialization to initialize a variable with a value it can not safely hold, the compiler will throw a warning or an error. For example:

```
1 | int width { 4.5 }; // error: not all double values fit into an int
```

In the above snippet, we're trying to assign a number (4.5) that has a fractional part (the .5 part) to an integer variable (which can only hold numbers without fractional parts). Copy and direct initialization would drop the fractional part, resulting in initialization of value 4 into variable *width*. However, with brace initialization, this will cause the compiler to issue an error (which is generally a good thing, because losing data is rarely desired). Conversions that can be done without potential data loss are allowed.

Value initialization and zero initialization

When a variable is initialized with empty braces, **value initialization** takes place. In most cases, **value initialization** will initialize the variable to zero (or empty, if that's more appropriate for a given type). In such cases where zeroing occurs, this is called **zero initialization**.

Initialize your variables

Initialize your variables upon creation. You may eventually find cases where you want to ignore this advice for a specific reason (e.g. a performance critical section of code that uses a lot of variables), and that's okay, as long the choice is made deliberately.

Initializing multiple variables

In the last section, we noted that it is possible to define multiple variables *of the same type* in a single statement by separating the names with a comma:

```
1 int a, b;
```

We also noted that best practice is to avoid this syntax altogether. However, since you may encounter other code that uses this style, it's still useful to talk a little bit more about it, if for no other reason than to reinforce some of the reasons you should be avoiding it.

You can initialize multiple variables defined on the same line:

```
int a = 5, b = 6; // copy initialization
int c( 7 ), d( 8 ); // direct initialization
int e { 9 }, f { 10 }; // brace initialization (preferred)
```

Unfortunately, there's a common pitfall here that can occur when the programmer mistakenly tries to initialize both variables by using one initialization statement:

```
int a, b = 5; // wrong (a is not initialized!)

int a = 5, b = 5; // correct
```

In the top statement, variable "a" will be left uninitialized, and the compiler may or may not complain. If it doesn't, this is a great way to have your program intermittently crash and produce sporadic results. We'll talk more about what happens if you use uninitialized variables shortly.

The best way to remember that this is wrong is to consider the case of direct initialization or brace initialization:

```
int a, b( 5 );
int c, d{ 5 };
```

Defining multiple variables

It is possible to define multiple variables *of the same type* in a single statement by separating the names with a comma. The following 2 snippets of code are effectively the same:

```
1 int a;
2 int b;
```

is the same as:

```
1 int a, b;
```

When defining multiple variables this way, there are two common mistakes that new programmers tend to make (neither serious, since the compiler will catch these and ask you to fix them):

The first mistake is giving each variable a type when defining variables in sequence.

```
1 int a, int b; // wrong (compiler error)
2
3 int a, b; // correct
```

The second mistake is to try to define variables of different types in the same statement, which is not allowed. Variables of different types must be defined in separate statements.

```
1 int a, double b; // wrong (compiler error)
2
3 int a; double b; // correct (but not recommended)
4
5 // correct and recommended (easier to read)
6 int a;
7 double b;
```