Introduction to Programming with C++

Data Types

Objective

- In this chapter we'll discuss:
  - Data types in C++
  - How you declare and initialize variables
  - What literals are and how you define them
  - Binary and hexadecimal integers
  - How calculations work
  - How you can fix the value of a variable
  - How to create variables that store characters
  - What the auto keyword does
  - What lvalues and rvalues are

Variables, Data, & Data Types

- A variable is a named piece of memory that you define.
- Each variable only stores data of a particular type.
- Every variable has a type that defines the kind of data it can store.
- Each fundamental type is identified by a unique type name that is a keyword.
- The compiler makes extensive checks to ensure that you use the right data type in any given context.
- It will also ensures compatibility during operations
- The compiler detects and reports attempts to combine data of different types that are incompatible.

Variables, Data, & Data Types

- Numerical values fall into two broad categories:
  - Integers-whole numbers in other words,
  - Floating-point values, which can be non-integral.
- There are several fundamental C++ types in each category,
- Each of these can store a specific range of values.

Defining Integer Variables

- Here's a statement that defines an integer variable:

```
int appleCount;
```

- This defines a variable of type int with the name `appleCount`.
- The variable will contain some arbitrary junk-value.
- You can and should specify an initial value when you define the variable, like this:

```
int appleCount {15}; // Number of apples
```

- The initial value for `appleCount` appears between the braces following the name so it has the value 15.
- The braces enclosing the initial value is called an initializer list.

Defining Integer Variables

- You'll meet situations later in the course where an initializer list will have several values between the braces.
- You don't have to initialize variables when you define them but it's a good idea to do so.
- Ensuring variables start out with known values makes it easier to work out what is wrong when the code doesn't work as you expect.
- Type int is typically 4 bytes, which can store integers from **-2,147,483,648** to **+2,147,483,647**.
- This covers most situations, which is why int is the integer type that is used most frequently.

Defining Integer Variables

- Here are definitions for three variables of type `int`:

```
int appleCount {15}; // Number of apples
int orangeCount {5}; // Number of oranges
int totalFruit {appleCount + orangeCount}; // Total number of fruit
```

- The initial value for `totalFruit` is the sum of the values of two variables defined previously.
- This demonstrates that the initial value for a variable can be an expression.

The statements that define the two variables in the expression for the initial value for totalFruit must appear earlier in the source file. Otherwise the definition for totalFruit won't compile.

Defining Integer Variables

- The initial value between the braces should be of the same type as the variable you are defining.
- If it isn't, the compiler will have to convert it to the required type.
- If the conversion is to a more limited range of values, it potentially could lead to loss of information
- Hence, the compiler won't convert the value but just flag it as an error.
- An example would be if you specified the initial value for an integer variable that is not an integer—1.5 for example.
- You might do this by accident when entering the value 15 for `appleCount`.

Defining Integer Variables

- A conversion to a type with a more limited range of values is called a narrowing conversion.
- There are two other ways for initializing a variable.
- Functional notation looks like this:

```
int orangeCount(5);
int totalFruit(appleCount + orangeCount);
```

- Alternatively you could write this:

```
int orangeCount = 5;
int totalFruit = appleCount + orangeCount;
```

Defining Integer Variables

- While both of these possibilities are valid
- It's recommended that you adopt the initializer list form.
- This is the most recent syntax that was introduced in C++ 11 to standardize initialization.
- It is preferred because it enables you to initialize just about everything in the same way.
- There’s one exception that we'll discuss later
- You can define and initialize more than one variable of a given type in a single statement.
- For example:

```
int footCount {2}, toeCount {10}, headCount {1};
```

- Most of the time it’s better to define each variable in a separate statement.
- This makes the code more readable and you can explain the purpose of each of them in a comment.

Signed Integer Types

Type Name	Typical Size (bytes)	Range of Values
`signed char`	1	-128 to 127
`short` `short int`	2	-256 to 255
`int`	4	4 -2,147,483,648 to +2,147,483,647
`long` `long int`	4	4 -2,147,483,648 to +2,147,483,647
`long long` `long long int`	8	8 -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

- Type signed char is typically 1 byte;
- The number of bytes occupied by the others depends on the compiler.
- Where two type names appear in the left column, the abbreviated name that comes first is commonly used
- So you will usually see long used rather than long int.
- Each type will have at least as much memory as the one that precedes it in the list.

Unsigned Integer Types

- There are circumstances where you don't need to store negative numbers.
- The number of students in a class or the number of parts in an assembly are always positive integers.
- You can specify integer types that only store non-negative values by prefixing any of the names of the signed integer types with the `unsigned` keyword
- For example
    - `unsigned char`
    - `unsigned short`
    - `unsigned long`

Unsigned Integer Types

- Each unsigned type is a different type from the signed type but occupies the same amount of memory.
- Type char is a different integer type from both signed char and unsigned char.
- Type char stores a character code and can be a signed or unsigned type depending on your compiler.
- If the constant `CHAR_MIN` in the `climits` header is 0, then char is an unsigned type with your compiler.

Unsigned Integer Types

- Here are some examples of variables of some of these types:

```
signed char ch {20};
long temperature {-50L};
long width {500L};
long long height {250LL};
unsigned int toe_count {10U};
unsigned long angel_count {1000000UL};
```

- Note how you write constants of type `long` and type `long long`.
- You must append L to the first and LL to the second.
- If there is no suffix, an integer constant is of type int.
- You can use lowercase for the L and LL suffixes but it can easily be confused with the digit 1.
- Unsigned integer constants have u or U appended.

Fixed Value Variables

- Sometimes you'll want to define variables with values that are fixed and must not be changed.
- You use the `const` keyword in the definition of a variable that must not be changed.
- For example:

```
const unsigned int toe_count {2U};
```

- The const keyword tells the compiler that the value of toe_count must not be changed.
- A statement that attempts to modify the value of toe_count will be flagged as an error during compilation.
- You can use the `const` keyword to fix the value of variables of any type.

Integer Literals

- Constants of any kind, such as `42`, or `2.71828`, `'Z'`, or `"Mark Twain"`, are referred to as literals.
- Every literal will be of some type.
- For example,
  - `42` is an integer literal,
  - `2.71828` is floating-point literal,
  - `‘Z'` is a character literal
  - `"Mark Twain"` is a string literal.

Integer Literals

- You can write integer literals in a very straightforward way.
- Here are some examples of decimal integers:

```
-123L +123 123 22333 98U -1234LL 12345ULL
```

- You have seen that unsigned integer literals have u or U appended.
- Literals of types long and type long long have L or LL appended respectively
- If they are unsigned, they also have u or U appended.
- The U and L or LL can be in either sequence.
- You could omit the + in the second example, as it’s implied by default
- But if you think putting it in makes things clearer, that’s not a problem.

Integer Literals

- The literal +123 is the same as 123 and is of type int because there is no suffix.
- The fourth example is the number that you would normally write as 22,333, but you must not use commas in an integer literal.
- Here are some statements using some of these literals:

```
unsigned long age {99UL}; // 99uL would be OK too
unsigned short {10u}; // There is no specific literal type for short
long long distance {1234567LL};
```

- You can't write just any old integer value as an initial value for a variable.
- An initializing value must be within the permitted range for the type of variable as well as match the type.
- A literal in an expression must be within the range of some type.

Hexadecimal Literals

- You can write integer literals as hexadecimal values.
- You prefix a hexadecimal literal with `0x` or `0X`
- `0x999` is a hexadecimal number of type int with three hexadecimal digits.
- Plain old `999`, on the other hand, is a decimal value of type int with decimal digits
- So the value will be completely different.
- Here are some more examples of hexadecimal literals:

Hexadecimal literals:	`0x1AF`	`0x123U`	`0xAL`	`0xcad`	`0xFF`
Decimal literals:	`431`	`291U`	`10L`	`3245`	`255`

Hexadecimal Literals

- A major use for hexadecimal literals is to define particular patterns of bits.
- Each hexadecimal digit corresponds to 4 bits so it's easy to express a pattern of bits as a hexadecimal literal.
- The red, blue, and green components (RGB values) of a pixel color are often expressed as three bytes packed into a 32-bit word.
- Here are some examples:

```
unsigned int color {0x0f0d0eU}; // decimal 986,382
int mask {0XFF00FF00}; // Four bytes specified as FF, 00, FF, 00
unsigned long value {0xdeadLU}; // decimal 57,005
```

Octal Literals

- You can write integer literals as octal value.
- You identify a number as octal by writing it with a leading zero.
- You can also write a binary integer literal as a sequence of binary digits (0 or 1) prefixed by 0b or 0B.

Octal literals:	657	0443U	012L	6255	377
Decimal literals:	431	291U	10L	3245	255
Binary Literals:	0B110101111	0b100100011U	0b1010L	0B11001101	0b11111111

- Binary literals were introduced by the C++ 14 standard
- As far as your compiler is concerned, it doesn't matter which number base you choose when you write an integer value.
- Ultimately it will be stored as a binary number.
- The different ways for writing an integer are there just for your convenience.
- You choose one or other of the possible representations to suit the context.

Calculations with Integers

- To begin with, let's get some bits of terminology out of the way.
- An operation such as addition or multiplication is defined by an operator
- The values that an operator acts upon are called operands
- For example, in an expression such as 2*3, the operands are 2 and 3.
- Operators such as multiplication that require two operands are called binary operators.
- Operators that require one operand are called unary operators.
- An example of a unary operator is the minus sign
- For example in the expression -width, the minus sign negates the value of width

Calculations with Integers

Operator	Operation	Example
+	Addition	4+5 is 9
-	Subtraction	4-5 is -1
*	Multiplication	4*5 is 20
/	Division	6/4 is 1 (integer division)
%	Modulus	6%4 is 2 (remainder of the devision)
=	Assignment	`long x = 57;` // the variable x will assume the value of 57 from this point on

More on Assignment Operations

- You can assign a value to more than one variable in a single statement.
- For example:

```
int a {}, b {}, c {5}, d{4};
a = b = c*c - d*d;
```

- The second statement calculates the value of the expression c*c-d*d and stores the result in b, so b will be set to 9.
- Next the value of b is stored in a so a will also be set to 9.
- You can have as many repeated assignments like this as you want.
- The operand on the left of an assignment can be a variable or an expression

More on Assignment Operations

- If it is an expression, the resultmust be an lvalue.
- An lvalue represents a persistent memory location so a variable is an lvalue.
- Every expression in C++ results in either an lvalue or an rvalue.
- An rvalue is a result that is not an lvalue, so it is transient.
- The result of the expression c*c-d*d in the statement above is an rvalue.
- The compiler allocates a temporary memory location to store the result of the expression
- Once the statement has been executed, the result and the memory it occupies is discarded.

- The difference between lvalues and rvalues is not important now
- But it will become very important when you delve into functions and classes.

The op= Assignment Operators

- In Ex2_01.cpp, there was a statement that you could write more economically:

```
feet = feet % feetPerYard;
```

- This statement could be written using an op= assignment operator.
- The `op=` assignment operators are so called because they're composed of an operator and an assignment operator `=`.
- You could write the previous statement as:

```
feet %= feetPerYard;
```

- This is exactly the same operation as the previous statement.
- In general, an op= assignment is of the form:

```
lhs op= rhs;
```

The op= Assignment Operators

- `lhs` represents a variable of some kind that is the destination for the result of the operator.
- `rhs` is any expression.
- This is equivalent to the statement:

```
lhs = lhs op (rhs);
```

- The parentheses are important because you can write statements such as:

```
x *= y + 1;
```

- This is equivalent to:

```
x = x*(y + 1);
```

- Without the implied parentheses, the value stored in `x` would be the result of `x*y+1`, which is quite different.
- You can use a range of operators for op in the `op=` form of assignment.

The op= Assignment Operators

<table>
<tr><th>Operation</th><th>Operator</th><th>Operation</th><th>Operator</th></tr>
<tr><td>Addition</td><td>+=</td><td>Bitwise AND</td><td>&=</td></tr>
<tr><td>Subtraction</td><td>-=</td><td>Bitwise OR</td><td>|=</td></tr>
<tr><td>Multiplication</td><td>*=</td><td>Bitwise exclusive OR</td><td>^=</td></tr>
<tr><td>Division</td><td>/=</td><td>Shift left</td><td><<=</td></tr>
<tr><td>Modulus</td><td>%=</td><td>Shift right</td><td>=>></td></tr>
</table>

- Note that there can be no spaces between op and the =.
- If you include a space, it will be flagged as an error.
- You can use += when you want to increment a variable by some amount.
- For example, the following two statements have the same effect:

`using` Declarations & Directives

- There were a lot of occurrences of `std::cin` and `std::cout` in Ex2_01.cpp.
- You can eliminate the need to qualify a name with the namespace name in a source file with a using declaration.
- Here's an example:

```
using std::cout;
```

- This tells the compiler that when you write `cout`, it should be interpreted as `std::cout`.
- This saves typing and makes the code look a little less cluttered.

`using` Declarations & Directives

- You can apply using declarations to names from any namespace, not just `std`.
- A using directive imports all the names from a namespace.
- Here's how you could use any name from the std namespace without the need to qualify it:

```
using namespace std; // Make all the names in std
available without qualification
```

- With this at the beginning of a source file, you don't have to qualify any name that is defined in the `std` namespace.

- At first sight this seems an attractive idea.
- The problem is it defeats a major reason for having namespaces.
- It is unlikely that you know all the names that are defined in std and with this using directive you have increased the probability of accidentally using a name from std.

The sizeof Operator

- You use the sizeof operator to obtain the number of bytes occupied by a type, or by a variable, or by the result of an expression.
- Here are some examples of its use:

```
int height {74};
std::cout << "The height variable occupies “
          << sizeof height << " bytes.”
          << std::endl;
std::cout << "Type \"long long\" occupies " <<
          sizeof (long long) << " bytes." <<
          std::endl;
std::cout << "The expression height * height/2”
          << occupies “ << sizeof (height*height/2)
          << " bytes." << std::endl;
```

The sizeof Operator

- These statements show how you can output the size of a variable, the size of a type, and the size of the result of an expression.
- To use sizeof to obtain the memory occupied by a type, the type name must be between parentheses.
- You also need parentheses around an expression with `sizeof`.
- You don't need parentheses around a variable name, but there's no harm in putting them in.
- The result that sizeof produces is of type `size_t`.
- `size_t` is an unsigned integer type that is defined in the Standard Library header `cstddef`.

Incrementing and Decrementing Integers

- You can use += and -= with the litral 1 on the rvalue to increment and decrement the lvalue by one
- There are two other operators that can perform the same tasks.
- They're called the increment operator and the decrement operators, ++ and -- respectively.
- These are unary operators that you can apply to an integer variable.
- The following statements that modify count have exactly the same effect:

```
int count {5};
count = count + 1;
count += 1;
++count;
```

Incrementing and Decrementing Integers

- Each statement increments count by 1.
- Using the increment operator is clearly the most concise.
- The effect in an expression is to increment the value of the variable and then to use the incremented value in the expression.
- For example, suppose count has the value 5, and you execute this statement:

```
total = ++count + 6;
```

- The increment and decrement operators execute before any other binary arithmetic operators in an expression.

Postfix Increment and Decrement Operations

- The postfix form of ++ increments the variable to which it applies after its value is used in context.
- For example, you can rewrite the earlier example as follows:

```
total = count++ + 6;
```

- With an initial value of 5 for count, total is assigned the value 11.
- count will then be incremented to 6.
- The preceding statement is equivalent to the following statements:

```
total = count + 6;
++count;
```

- These rules also apply to the decrement operator.

Postfix Increment and Decrement Operations

- You must not apply the prefix form of these operators to a given variable more than once in an expression.
- Suppose count has the value 5, and you write this:

```
total = ++count * 3 + ++count * 5;
```

- Because the statement modifies the value of count more than once, the result is undefined.
- You should get an error message from the compiler with this statement.
- Note also that the effects of statements such as the following are undefined:

```
k = ++k + 1;
```

- Although such expressions are undefined according to the C++ standard, this doesn't mean that your compiler won't compile them.
- It just means that there is no guarantee of consistency in the results.

Defining Floating-Point Variables

- You use floating-point variables whenever you want to work with values that are not integral.
- There are three floating-point data types.
    - `float` - Single precision floating-point values
    - double - Double precision floating-point values
    - long double - Double-extended precision floating-point values
- The term “precision” refers to the number of significant digits in the mantissa.
- The types are in order of increasing precision

Defining Floating-Point Variables

- Note that the precision only determines the number of digits in the mantissa.
- The range of numbers that can be represented by a particular type is determined by the range of possible exponents.
- The precision and range of values aren't prescribed by the C++ standard so what you get with each type depends on your compiler.
- This will depend on what kind of processor is used by your computer and the floating-point representation it uses.
- Type long double will provide a precision that's no less than that of type double, and type double will provide a precision that is no less than that of type float.

Defining Floating-Point Variables

- Typically,
  - `float` provides 7 digits precision
  - `double` provides 15 digits,
  - `long double` provides 19 digits precision
  - `double` and `long double` have the same precision with some compilers.
- Typical ranges of values on an Intel processor are

Type	Precision	Range
`float`	7	**1.2×10^{-38} to 3.4×10^{38}**
`double`	15	**2.2×10^{-308} to 1.8×10^{308}**
`long double`	19	**3.3×10^{-4932} to 1.2×10^{4932}**

Floating-Point Literals

- You can see from the code fragment in the previous section that float literals have f (or F) appended and long double literals have L (or l) appended.
- Floating point literals without a suffix are of type double.
- A floating-point literal includes either a decimal point, or an exponent, or both
- A numeric literal with neither is an integer.
- An exponent is optional in a floating-point literal
- It represents a power of 10 that multiplies the value.
- An exponent must be prefixed with e or E and follows the value.
- Here are some floating-point literals that include an exponent:

  `5E3` (5000.0) `100.5E2` (10050.0) `2.5e-3` (0.0025) `-0.1E-3L` (-0.0001L) `.345e1F` (3.45F)

- Exponents are particularly useful when you need to express very small or very large values.

Floating-Point Calculations

- You write floating-point calculations in the same way as integer calculations.
- For example:

```
const double pi {3.1414926};
double a {0.75}; // Thickness of a pizza
double z {5.5}; // Radius of a pizza
double volume {}; // Volume of pizza - to be calculated
volume = pi*z*z*a;
```

- The modulus operator, %, can’t be used with floating-point operands.
- You can also apply the prefix and postfix increment and decrement operators

Reading Assignment

- Mathematical Functions (page 40)
- Formatting Stream Output(Page 43)