# ■Chapter 2: Introducing Fundamental Types of Data 27

## Variables, Data, and Data Types 27

- **Variable** is a named piece of memory that you define. Each variable stores data only 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 consists of one or more **keywords**.
- Keywords are reserved words in C++ that you cannot use for anything else.
- Numerical values fall into two categories: **integers**, which are whole numbers, and **floating-point values**, which can be nonintegral.

## Defining Integer Variables 28

- The braces enclosing the initial value are called a **braced initializer**, can have Several Values.
- A conversion to a type with a more limited range of values is called a **narrowing conversion**.
- **Functional notation** looks like this: int orange_count(5); int total_fruit(apple_count + orange_count);
- alternative is the **so-called assignment notation**: int orange_count = 5; int total_fruit = apple_count + orange_count;
- **braced initializer syntax** was introduced in C++11 specifically to standardize initialization. Its main advantage is that it enables you to initialize just about everything in the same way—which is why it is also commonly referred to as **uniform initialization**.

```
int banana_count(7.5);        // May compile without warning
int coconut_count = 5.3;      // May compile without warning
int papaya_count{0.3};        // At least a compiler warning, often an error
```

| SIGNED INTEGER TYPES | | |
|---|---|---|
| **Type Name** | **Typical Size (Bytes)** | **Range of Values** |
| **(LETTERS)** signed char | 1 | -128 to +127 |
| **(WHOLE NUMBERS)** short<br>short int<br>signed short<br>signed short int | 2 | -256 to +255 |
| **(WHOLE NUMBERS)** int<br>signed<br>signed int | 4 | -2,147,483,648 to +2,147,483,647 |
| **(WHOLE NUMBERS)** long<br>long int<br>signed long<br>signed long int | 4 or 8 | Same as int or long long |

| | | |
|---|---|---|
| **(WHOLE NUMBERS)** long long<br>long long int<br>signed long long<br>signed long long int | 8 | 9,223,372,036,854,775,808 to<br>+9,223,372,036,854,775,807 |

## Zero Initialization 31

int counter {0}; // counter starts at zero = int counter {}; // counter starts at zero

**Zero initialization** works for any fundamental type. For all fundamental numeric types, for instance, an empty braced initializer is always assumed to contain the number zero.

## Defining Variables with Fixed Values 32

**const** keyword in the definition of a variable that must not be changed. Such variables are often referred to as **constants**.

example: *const unsigned toe_count {10};* // An unsigned integer with fixed value 10

■ Tip If nothing else, knowing which variables can and cannot change their values along the way makes your code easier to follow. So, we recommend you add the const specifier whenever applicable.

## Integer Literals 32

Constant values of any kind, such as 42, 2.71828, 'Z', or "Mark Twain", are referred to as **literals**. These examples are, in sequence, an **integer literal**, a **floating-point literal**, a **character literal**, and a **string literal**.

## Decimal Integer Literals 32

examples of decimal integers: -123L +123 123 22333 98U -1234LL 12345ULL

Unsigned integer literals have u or U appended. Literals of types long and type long long have L or LL appended, respectively, and if they are unsigned, they also have u or U appended

```
unsigned long age {99UL}; // 99ul or 99LU would be OK too
unsigned short price {10u}; // There is no specific literal type for short
long long distance {15'000'000LL}; // Common digit grouping of the number 15 million
```

■ Note While mostly optional, there are situations where you do need to add the correct literal suffixes, such as when you initialize a variable with type auto (as explained near the end of this chapter) or when calling overloaded functions with literal arguments

## Hexadecimal Literals 33

unsigned int color {0x0f0d0e}; // Unsigned int hexadecimal constant - decimal 986,382 int mask {0XFF00FF00}; // Four bytes specified as FF, 00, FF, 00 unsigned long value {0xDEADlu}; // Unsigned long hexadecimal literal - decimal 57,005

## Octal Literals 34 Out-of-date

## Binary Literals 34

- binary integer literal as a sequence of binary digits (0 or 1) prefixed by either 0b or 0B.
- binary literal can have L or LL as a suffix to indicate it is type long or long long, and u or U if it is an unsigned literal.

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| Binary literals: | 0B110101111 | 0b100100011U | 0b1010L | 0B110010101101 | 0b11111111 |
| Decimal literals: | 431 | 291U | 10L | 3245 | 255 |

## Calculations with Integers 35

- An operation, addition or multiplication is defined by an **operator**—for addition and multiplication are + and *, respectively.
- The values that an operator acts upon are called **operands**, so in an expression such as 2*3, the operands are 2 and 3.
- Need 2 Operators - **binary operators**. Need 1 Operators - **unary operators**.

| OPERATOR | OPERATION |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modulus (the remainder after division) |

The modulus operator, %, complements the division operator in that it produces the remainder after integer division. It is defined such that, for all integers x and y, **(x / y) * y + (x % y) == x**

## Compound Arithmetic Expressions 36

Parenthesis arithmetic comes first.
multiplication / division / modulus operations / addition and subtraction.
Nested Parenthesis: 2*(a + 3*(b + 4*(c + 5*d)))

## Assignment Operations 37

This last line is an **assignment statement**, and the = is the **assignment operator**.

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

Variables can be overwritten as many times as you want, except constants, of course.

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

## The op= Assignment Operators 40

```
feet = feet % feet_per_yard;
```

This statement could be written using an op= **assignment operator**. The op= assignment operators, or also **compound assignment operators**, are so called because they're composed of an operator and an assignment operator =. You could use one to write the previous statement as follows:

```
feet %= feet_per_yard;
```

x *= y + 1; ---------- This is equivalent to the following:
x = x * (y + 1);

| op= Assignment Operators | | | |
|---|---|---|---|
| **Operation** | **Operator** | **Operation** | **Operator** |
| Add | += | Bitwise AND | &= |
| Subtract | -= | Bitwise OR | \|= |
| Multiply | *= | Bitwise exclusive OR | ^= |
| Division | /= | Shift Left | <<= |
| Modulus | %= | Shift Right | >>= |

**DECLARATIONS AND DIRECTIVES**

You can eliminate the need to qualify a name with the namespace name in a source file with a ***using*** *declaration*.

With this declaration before the main() function definition, you can write cout instead of std::cout

You could include two using declarations at the beginning of Ex2_02.cpp and avoid the need to qualify cin and cout:

**using std::cin;**
**using std::cout;**

**using std::endl;**

## The sizeof Operator 42

use the **sizeof operator** to obtain the number of bytes occupied by a type, 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 result of the expression height * height/2 occupies "
 << sizeof(height * height/2) << " bytes." << std::endl;
```

## Incrementing and Decrementing Integers 42

- **increment operator** and the **decrement operator**, **++** and **--**
- The following three statements that modify **count** have exactly the same effect:

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

- ++ or -- operator before the variable is called the **prefix form.** Incrementation happens first.
- You can also place them after a variable, which is called the **postfix form**. Incrementation happens last.

## Postfix Increment and Decrement Operations 43

## Defining Floating-Point Variables 44

| Data Type | Description | Example |
|---|---|---|
| float | Single precision floating-point values | 2.00000000 |
| double | Double precision floating-point values | 2.0000235 |
| long double | Double-extended precision floating-point values | |

**precision** refers to the number of significant digits in the mantissa.

| Type | Precision (Decimal Digits) | Range (+ or –) |
|---|---|---|
| float | 7 | $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$ |
| double | 15 (nearly 16) | $\pm 2.22 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$ |
| long double | 18-19 | $\pm 3.65 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$ |

```
float pi {3.1415926f}; // Ratio of circle circumference to diameter
double inches_to_mm {25.4};
long double root2 {1.4142135623730950488L}; // Square root of 2
```

## Floating-Point Literals 45

- 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 and 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)

## Floating-Point Calculations 46

```
const double pi {3.141592653589793};      // Circumference of a circle divided by its diameter
double a {0.2};        // Thickness of proper New York-style pizza (in inches)
double z {9};          // Radius of large New York-style pizza (in inches)
double volume {};      // Volume of pizza - to be calculated
volume = pi*z*z*a;
```

- The modulus operator, %, can't be used with floating-point operands, but all the other binary arithmetic
- operators that you have seen, +, -, *, and /, prefix and postfix increment and decrement operators, ++ and -- can be used

## Pitfalls (Of Floating-Point Values)46

- Many decimal values don't convert exactly to binary floating-point values. The small errors that occur can easily be amplified in your calculations to produce large errors.

- Taking the difference between two nearly identical values will lose precision. If you take the difference between two values of type float that differ in the sixth significant digit, you'll produce a result that will have only one or two digits of accuracy. Resulting in catastrophic cancellation.
- Working with values that differ by several orders of magnitude can lead to errors. An elementary example of this is adding two values stored as type float with 7 digits of precision where one value is 108 times larger than the other. You can add the smaller value to the larger as many times as you like, and the larger value will be unchanged.

## Invalid Floating-Point Results 47

the result of division by zero is undefined.

binary mantissa of all zeroes and an exponent of all ones to represent +infinity or -infinity, depending on the sign. When you divide a positive nonzero value by zero, the result will be **+infinity**, and dividing a negative value by zero will result in **-infinity**.

Another special floating-point value defined by this standard is called **not-a-number**, or **NaN**. This represents a result that isn't mathematically defined, such as when you divide zero by zero or infinity by infinity.

| Operation | Result | Operation | Result |
|---|---|---|---|
| ±value / 0 | ±infinity | 0 / 0 | NaN |
| ±infinity ± value | ±infinity | ±infinity / ±infinity | NaN |
| ±infinity * value | ±infinity | infinity - infinity | NaN |
| ±infinity / value | ±infinity | infinity * 0 | NaN |

To check whether a given number is either infinity or NaN, you should use the **std::isinf()** and **std::isnan()** functions provided by the **#include<cmath>** header

## Mathematical Functions 48

### #include<cmath> ---→ std::FUNCTION

Standard Library header file defines a large selection of trigonometric and numerical functions that you can use in your programs.

| FUNCTION | DESCRIPTION |
|---|---|
| **abs(arg)** | computes **Absolute Value** of an Argument. Unlike most cmath functions, abs() returns an integer type if the arg is integer. |
| **ceil(arg)** | (Searches for ceiling) computes from the **smallest floating-point value**. ceil <= arg<br>EXAMPLE: std::ceil(2.5) produces 3.0 and std::ceil(-2.5) produces -2.0. |
| **floor(arg)** | (Searches for floor) computes from the **largest floating-point value**. floor >= arg<br>EXAMPLE: std::floor(2.5) results in 2.0 and std::floor(-2.5) results in -3.0 |
| **exp(arg)** | computes the value of **EArg** |
| **log(arg)** | Computes the **Natural Logarithm (to base e)** of arg. |
| **log10(arg)** | computes the **Logarithm to base 10** of arg |
| **pow(arg1, arg2)** | **$\text{Arg1}^{\text{Arg2}}$** . can be integer or floating-point types.<br>std::pow(2, 3) is 8.0 , std::pow(1.5f, 3) equals 3.375f , and std::pow(4, 0.5) is equal to 2. |
| **sqrt(arg)** | **Square Root** of Arg |

| | |
|---|---|
| **round(arg)** | **Round** Arg to nearest int(floating-point[Whole] number). std::round(-1.5f) gives -2.0f |
| **lround() & llround()** | **Round** to int of type long & long long. std::lround(0.5) gives 1L |
| **cos(), sin(), and tan())** inverse functions-> **(std::acos(), asin(), and atan())** | **Angles** are always expressed in **radians** |

calculate the **Cosine of an angle in radians:**

*double angle {1.5}; // In radians*
*double cosine_value {std::cos(angle)};*

If the angle is in degrees, you can calculate the **Tangent by using a value for π to convert to radians:**

*float angle_deg {60.0f}; // Angle in degrees*
*const float pi { 3.14159265f };*
*const float pi_degrees {180.0f};*
*float tangent {std::tan(pi * angle_deg/pi_degrees)};*

If you know the height of a church steeple is 100 feet and you're standing 50 feet from its base, you can **calculate Tangent the angle in radians of the top of the steeple like this:**

*double height {100.0}; // Steeple height- feet*
*double distance {50.0}; // Distance from base*
*double angle {std::atan(distance / height)}; // Result in radians*

You can use this value in angle and the value of distance to **calculate the distance from your toe to the top of the steeple:**

*double toe_to_tip {distance / std::sin(angle)};*

## Formatting Stream Output 51

You can change how data is formatted when it is written to an output stream using **stream manipulators**, which are declared in the **#include<iomanip>** and **#include<ios>** Standard Library headers.

You apply a stream manipulator to an output stream with the insert operator, <<. We'll just introduce the most useful manipulators.

Consult a Standard Library reference if you want to get to know the others.

All manipulators declared by ios are automatically available if you include the familiar **#include<iostream>** header. Unlike those of the **#include<iomanip>** header, these stream manipulators do not require an argument:

| Stream Manipulator | Description |
|---|---|
| **std::fixed** | Output: Floating-point data in a **fixed-point notation** |
| **std::scientific** | Output: All Subsequent Floating-point data in **Scientific Notation**, which always includes an exponent and one digit before the decimal point. |
| **std::defaultfloat** | Revert to **default floating-point** data presentation. |
| **std::dec** | All subsequent int outputs are **Decimal**. |
| **std::hex** | All subsequent int outputs are **Hexadecimal**. |
| **std::oct** | All subsequent int outputs are **Octal**. |
| **std::showbase** | Outputs the **base prefix for hexadecimal and octal integer values**. Inserting **std::noshowbase** in a stream will switch this off. |

| | |
|---|---|
| **std::left** | Output **justified to the left**. |
| **std::right** | Default. Output **justified to the right**. |
| **#include<iomanip>** (header)->Stream Manipulators | |
| **std::setprecision(n)** | Sets the floating-point precision or the number of decimal places to n digits. **OR** If the default floating-point output presentation is in effect, n specifies the number of digits in the output value. **OR** If fixed or scientific format has been set, n is the number of digits following the decimal point. The default precision is 6. |
| **std::setw(n)** | Sets the output field width to n characters, but only for the next output data item. Subsequent output reverts to the default where the field width is set to the number of output character needed to accommodate the data. |
| **std::setfill(ch)** | When the field width has more characters than the output value, excess characters in the field will be the default fill character, which is a space. This sets the fill character to be ch for all subsequent output. |

## Mixed Expressions and Type Conversion 53

**implicit conversions** -- compiler will arrange to convert one of the operand values to the same (Data) type as the other.

compiler chooses the operand with the type that has the more limited range of values as the one to be converted to the type of the other. In effect, it ranks the types in the following sequence, from high to low:

1. long double 2. double 3. float
4. unsigned long long 5. long long 6. unsigned long
7. long 8. unsigned int 9. int

subtraction of unsigned integers wraps around to very large positive numbers is sometimes called **underflow**.

the converse phenomenon exists as well and is called **overflow**.

## Explicit Type Conversion 54

### static_cast<type_to_convert_to>(expression)

The **static_cast** keyword reflects the fact that the cast is checked statically, that is, when the code is compiled. With classes, the conversion is checked **dynamically**, that is, when the program is executing.

By adding an explicit cast, you signal the compiler that a narrowing conversion is intentional.

With integer division, casting from a floating-point type to an integral type uses **truncation**.

The std::round(), lround(), and llround() functions from the cmath header allow you to round floating-point numbers to the nearest integer. In many cases, this is better than (implicit or explicit) casting, where truncation is used instead.

## Old-Style Casts 56 Don't Use. Outdated C command for C++ (type_to_convert_to)expression

## Finding the Limits 57

To display the maximum value you can store in a variable of type double, you could write this:

**std::cout << "Maximum value of type double is " <<
std::numeric_limits<double>::max();**

To get the lowest negative value a type can represent, you should use lowest() instead:

**std::numeric_ limits::lowest()**

## Finding Other Properties of Fundamental Types 58

You can retrieve many other items of information about various types. The number of binary digits, or bits for example, is returned by this expression: **std::numeric_limits<type_name>::digits**

To obtain the special floating-point values for infinity and not-anumber (NaN), you should use expressions of the following form:

**float positive_infinity = std::numeric_limits<float>::infinity();**
**double negative_infinity = -std::numeric_limits<double>::infinity();**
**long double not_a_number = std::numeric_limits<long double>::quiet_NaN();**

Besides quiet_NaN(), there's a function called signaling_NaN()

## Working with Character Variables 59

Variables of type **char** are used primarily to store a code for a single character and occupy 1 byte.

**char letter;** // Uninitialized - so junk value
**char yes {'Y'},no {'N'};** // Initialized with character literals
**char ch {33};** // Integer initializer equivalent to '!'

You can initialize a variable of type char with a character literal between single quotes or by an integer.

```
char ch {'A'};
char letter {ch + 5};     // letter is 'F'
++ch;                     // ch is now 'B'
ch += 3;                  // ch is now 'E'
```

When you write a char variable to cout, it is output as a character, not as an integer.
If you want to see it as a numerical value, you can cast it to another integer type. Here's an example:

```
std::cout << "ch is '" << ch
   << "' which is code " << std::hex << std::showbase
   << static_cast<int>(ch) << std::endl;
```

This produces the following output:

```
ch is 'E' which is code 0x45
```

## Working with Unicode Characters 60

4 integer types that store Unicode characters: **char, wchar_t, char16_t,** and **char32_t**

**Unicode** –Gives Ability to work with characters for multiple Languages simultaneously or if you want to handle character sets

for many non-English languages, 256 character codes doesn't go nearly far enough. Int Code for Letters.

Type **wchar_t** is a fundamental type that can store all members of the largest extended character set that's supported by an implementation. The type name derives from wide characters because the character is "wider" than the usual single-byte character. By contrast, type char is referred to as "narrow" because of the limited range of character codes that are available.

You define wide-character literals in a similar way to literals of type char, but you prefix them with L. Here's an example:

**wchar_t wch {L'Z'};**

---

This defines wch as type wchar_t and initializes it to the wide-character representation for Z. Your keyboard may not have keys for representing other national language characters, but you can still create them using hexadecimal notation. Here's an example:

**wchar_t wch {L'\x0438'}; // Cyrillic и**

---

**single quotes** is an escape sequence that specifies the hexadecimal representation of the character code.

Type wchar_t does not handle international character sets very well. It's much better to use type char16_t, which stores characters encoded as UTF-16, or char32_t, which stores UTF-32 encoded characters. Here's an example of defining a variable of type char16_t:

**char16_t letter {u'B'}; // Initialized with UTF-16 code for B**

**char16_t cyr {u'\x0438'}; // Initialized with UTF-16 code for cyrillic и**

---

The lowercase u prefix to the literals indicates that they are UTF-16. You prefix UTF-32 literals with **uppercase U**. Here's an example:

**char32_t letter {U'B'};** // Initialized with UTF-32 code for B

**char32_t cyr {U'\x044f'};** // Initialized with UTF-32 code for cyrillic я

---

Of course, if your editor and compiler have the capability to accept and display the characters, you can define cyr like this:

**char32_t cyr {U'я'};**

---

You should not mix output operations on wcout with output operations on cout.

## The auto Keyword 61

Compiler Automatically assigns Data Types to a Variable.

**auto m {10};** // m has type int

**auto n {200UL};** // n has type unsigned long

**auto pi {3.14159};** // pi has type double

---

-The compiler will deduce the types for m, n, and pi from the initial values you supply.

-You can use functional or assignment notation with auto for the initial value as well:

**auto m = 10;** // m has type int

**auto n = 200UL;** // n has type unsigned long

**auto pi(3.14159);** // pi has type double

---

you should simply never use braced initializers with auto.

Instead, either explicitly state the type or use assignment or functional notation.

## Summary 62

- Constants of any kind are called literals. All literals have a type.
- You can define integer literals as decimal, hexadecimal, octal, or binary values.
- A floating-point literal must contain a decimal point or an exponent or both. If there is neither, you have specified an integer.
- The fundamental types that store integers are short, int, long, and long long. These store signed integers, but you can also use the type modifier unsigned preceding any of these type names to produce a type that occupies the same number of bytes but stores unsigned integers.
- The floating-point data types are float, double, and long double.
- Uninitilized variables generally contain garbage values. Variables may be given initial values when they're defined, and it's good programming practice to do so. A braced initializer is the preferred way of specifying initial values.
- A variable of type char can store a single character and occupies one byte. Type char may be signed or unsigned, depending on your compiler. You can also use variables of the types signed char and unsigned char to store integers. Types char, signed char, and unsigned char are different types.
- Type wchar_t stores a wide character and occupies either two or four bytes, depending on your compiler. Types char16_t and char32_t may be better for handling Unicode characters in a cross-platform manner.
- You can fix the value of a variable by using the const modifier. The compiler will check for any attempts within the program source file to modify a variable defined as const.
- The four main mathematic operations correspond to the binary +, -, *, and / operators.

For integers, the modulus operator % gives you the remainder after integer division.

- The ++ and -- operators are special shorthand for adding or subtracting one in numeric variables. Both exist in postfix / prefix forms.
- You can mix different types of variables and constants in an expression. The compiler will arrange for one operand in a binary operation to be automatically converted to the type of the other operand when they differ.
- The compiler will automatically convert the type of the result of an expression on the right of an assignment to the type of the variable on the left where these are different. This can cause loss of information when the left-side type isn't able to contain the same information as the right-side type—double converted to int, for example, or long converted to short.
- You can explicitly convert a value of one type to another using the static_cast<>() operator.