Chapter

3

Numbers & Expressions

C++ comes with a wide variety of built in numeric types. There are signed and unsigned integers in five different sizes, as well as three different sizes of floating-point (or real) numbers. In addition, the standard library contains a complex number class, and it is easy to create your own custom numeric types.

Integers

Integers are whole numbers; the name is Latin, meaning "whole". Mathematical integers are infinite, but the C++ varieties are **finite**; each stored in a **fixed region of memory**.

The sizes for C++ integers are: `short`, `int`, `long`, and `long long`. C++ **does not** specify an **exact range** or **representation** for the integers. Both are **implementation dependent**.

- Size **cannot decrease** as you move from `short` to `int` to `long` to `long long`.
- `int` must use at least 2 bytes (16 bits), `long` must use at least 4 bytes (32 bits), and `long long` must use at least 8 bytes of storage (64 bits).

On our platform, `int` is 32 bits, `long` and `long long` are both 64 bits. On other platforms (such as Visual C++), `long` is 32 bits, just like the `int`.

Signed and Unsigned

C++ integers come in two "flavors": **signed** and **unsigned**. Unsigned variables offer **twice the range of positive values**, but cannot store negative numbers. For example, a 32-bit `int` has a maximum value of `2,147,483,647`, while the maximum `unsigned int` is `4,294,967,295`. C++ allows `unsigned int` to be abbreviated to `unsigned`.

Literals

Explicit values like `235` or `-75` are called **literals**. Integer literal are a sequence of decimal digits, with no spaces or commas allowed, preceded by an optional (+/-) sign. It is stored as a **signed `int`.** You may change the representation from `signed` to `unsigned` by add a `U` to the end. You may change the storage from `int` to `long`, or to `long long` by adding an `L` or an `LL`. Here are some examples.

```
1  auto a = 15;    // signed decimal int
2  auto b = 15L;   // signed decimal long
3  auto c = 15LL;  // signed decimal long long
4  auto d = 15UL;  // unsigned decimal long
```

Integer literals.

You can also write literals in base 8 (**octal**), base 16 (**hexadecimal**) and base 2 (**binary**).

```
auto oct32 = 040;          // 4 8s and no 0s
auto hex32 = 0x20;         // 2 16s and no 0s
auto bin32 = 0b10'0000;    // 1 32 and no 16s, 8s, 4s, 2s or 1s
```

Using **auto** instead of an explicit type to create the variables **a**, **b**, **c**, and **d**, allows the compiler to **infer** types from the initializers. **Type inference** is a new feature of C++11.

Floating-point Numbers

Numbers with a decimal fraction are called **floating-point numbers**. They are used to **model real numbers** from mathematics. C++ has three different floating-point types: **float**, **double**, and **long double**.

> *Generally, use double, not float or long double.*

Floating-point literals in C++ are written in two ways:

- Using **fixed-point notation** (**2.0**). The value is stored as a **double**.
- Using scientific or exponential notation (**2.9979E+8** to represent the speed of light, instead of writing it as **299790000**.) The exponent can be positive or negative and you can use an uppercase or lowercase "**E**".

You can change the **storage** of your literals by appending an **F** for type **float** and an **L** for a type **long double**. Most of the time you will ignore that.

Floating-point Output

The C++ output objects display floating-point numbers by choosing the representation that is most compact, limiting the default number of digits to **6**.

To **explicitly** set the output format involves 3 steps (only once in your program):

1. Add **#include <iomanip>** to the list of libraries you are using.
2. Send the **fixed** manipulator to the stream before printing any floating-point.

3. **Specify the decimal places** to be displayed, using **`setprecision(n)`**.

Here's an example, displaying the **`double`** variable **`cost`** with **two digits** of precision:

```
cout << fixed << setprecision(2) << cost;
```

When printing numbers, you may want to line up the decimal points correctly, so that the output is easier to read. To do that, use **`setw(width)`** where **`width`** is the width of the column that you want to display. Unlike **`setprecision()`**, **`setw()`** only applies to **one output object**. Here's an example:

```
cout << fixed << setprecision(2); // once (persistent)
cout << "Widget cost: " << setw(10) << cost << endl;
cout << "Sales price: " << setw(10) << price << endl;
```

Expressions & Calculations

To perform calculations, you write expressions to calculate the answer in a form similar to that used in mathematics. Consider the quadratic equation:

$$ax^2 + bx + c = 0$$

This equation has two solutions given by the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

To solve this in C++, you write an **expression** which uses **+** in place of the **±** symbol, to calculate one of the roots, like this:

```
1 | (-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

A C++ expression.

An Expression Vocabulary

An **expression** is any combination of **operators** and **operands** which yields a value.

- An **operand** indicates a **value**. Operands include:
  - → **Literals:** which represent a value
  - → **Variables:** a storage location containing a value
  - → **Function calls:** which can produce a value
  - → **Sub-expressions:** which compute a value

- An **operator** is a symbol which performs an operation on one or more operands and, subsequently, produces a value. Operators have three characteristics:
  - → **Arity**: the number of operands required. **Unary** operators require a single operand, **binary** operators require two.
  - → **Precedence**: determines which operands "bind to" the operator. Those with **higher precedence** "stick to" their adjacent operands more closely.
  - → **Associativity**: determines whether operations **at the same level of precedence** should proceed from right-to-left, (called **right-associative**), or from left-to-right, (called **left-associative**).

This linked table shows the precedence and associativity for all of the C++ operators.

Expression Evaluation

When operators and operands are **evaluated**, each operator is applied to its operands, and a **temporary value** is calculated. This is the **result** or **value** of the expression.

```
1 | cout << 10 + 1 << endl;
```

Expression evaluation.

Let's see how this is **evaluated**:

1. **Four operands**: the objects **`cout`** and **`endl`**, and two **`int`** literals, along with **three operators**: the addition operator and two instances of **`<<`**.
2. Addition has higher precedence, so the first result is the temporary value **`11`**.
3. Now we have three operands and two operators at the same precedence. Fall back on associativity (left to right) and evaluate **`cout << 11`**.
4. This expression has a side effect (it prints the value **`11`**), but it also produces a result, which is the **`cout`** object. The last expression is **`cout << endl`**;

See how this works by using parentheses, like this:

```
1 | cout << 10 + 1 << endl;
2 | cout << (10 + 1) << endl; // Precedence
3 | (cout << 11) << endl;     // Associativity, print 11
4 | cout << endl;             // Side effect (newline)
```

Expression evaluation.

In C and C++, the **order of operation** (specified by precedence and associativity) and the **order of evaluation** are not identical. Here's a simple example:

```
1 | x = a() * b() + c();
```

Order of evaluation.

Order of operation guarantees that the results of `(a() * b())` will be calculated before the addition of `c()`. However **no guarantees** are made about the order the functions will be called in: `c()` could be called first or `a()` could be called first.

If functions have no side effects (**idempotent functions**) this doesn't make a difference. If functions have side effects, such as printing, the result is **undefined**.

Side Effects & Functions

Most of you are familiar with expressions involving addition, subtraction, multiplication and division from Java or Python. However, when it comes to C++ you'll find a few surprises. We want to start this lesson by discussing the differences between **integer division** and normal or **true division**.

Then, we want to take a closer look at assignment and other **side-effect operators**, such as increment and decrement. We'll finish by talking about evaluating expressions involving **different types of operands**., and the C++ standard **mathematical functions**.

Integer Division and Remainders

Integer division works like grade-school long division. You draw a little "house" on the board and put the number you want to divide (called the **dividend**) inside the house. You draw the number you want to divide by (the **divisor**), standing at the front door of the house like a visitor. You then ask, "how many visitors" could fit inside the house and place that number on the roof. This is the **quotient**.

You multiply the quotient by the divisor, place the result beneath the dividend, and subtract. The **remainder** is anything left over (down in the "basement"), **8** in the example the student is solving on the board (on the left), and **3** in the example on the callout.

In C++ **integer division**, the quotient is calculated, and then **truncated** (not rounded). The remainder is **discarded**. With **true division,** `15/4` would be `3.75` but with integer division, it's just `3`, not `4` as it would be if the `3.75` was rounded.

The `%` or **remainder operator** (sometimes called the **modulus** operator) does exactly the same thing, except, instead of returning the quotient portion from the roof, it **returns the remainder** from the basement.

Side-Effect Operators

With the expression **cout << 11**, the **cout** object is changed and the character pair **11** appears on the screen. The change is a **side effect**. Here are other side-effect operators.

Chained Assignment

When using the assignment operator, the **result or value** of the expression is the value that is copied. Because **assignment is right associative**, we can "chain" assignment statements together like this:

```
1  int x, y, z;
2  x = y = z = 10;       // Chained assignment
3  x = y = (z = 10);     // Right associative
4  x = (y = 10);
5  x = 10;
```

Chained assignments.

Shorthand Assignment

To **modify an existing variable**, the shorthand-assignment operators let you do that.

```
x += 5;  // x = x + 5
```

Increment and Decrement

To **add or subtract one** from a variable use **increment** (**++**) and **decrement** (**--**). These are **unary operators** that can **only** be applied to a variable (***Lvalue***).

```
1  int a = 5, b = 10;
2  a++;          // a is changed to 6
3  --b;          // b is changed to 9
```

Increment and decrement statements.

In addition to the side effect, these operators produce a value. When placed **before** a variable, it is called **pre-increment** (or decrement);when placed **after**, it is called a **post-increment** (or decrement) expression. The side effect is the same for both: the variable is left with a value one greater (or less) than it was before.

The **expression value** (result) produced depends on whether it is post or pre-increment.

```
1 int a = 5, b = 10, c, d;
2 c = a++;        // a is changed to 6; c is assigned 5
3 d = --b;        // b is changed to 9 and so is d
```

Pre/post increment and decrement values.

With **pre-increment**, the variable is **first modified** and the **modified variable** is returned as the value. A prefix expression is thus an ***lvalue***, so `++++a` is legal.

With **post-increment**, the original value is saved to a **temporary**. Then, the variable is changed. Finally, the temporary is returned from the expression. That's why `c` is given the value `5`, and not `6`. A postfix expression is an ***rvalue***.

A Side-effect Pitfall

Don't ever use any side-effect operator twice in the same expression, These expressions all result in **undefined behavior**. Try it in g++, visual c++ and in clang++.

```
1 int n = 6;
2 print(n, ++n);  // 6,7 or 7,7?
3 int a = n * n++;
4 n = n++;
5 cout << n++ << n++ << n++ << endl;
```

Undefined side-effect values.

Mixed Expressions & Type Casts

Every expression produces a value, and each value produced **has a particular type**. Thus, when you add or subtract two integers, the **result** is an integer. But what if....

```
1 a = 5 * 3.5;
```

Mixed-type expressions.

The CPU uses **different circuitry** for integer and floating-point calculations. To evaluate this expression, **both operands** must be type `int` **or** they both must be type `double`. If we convert both to `int`, we **lose information**; converting them to `double` does not.

When your compiler encounters an expression that uses different types, it determines the operand with the greatest **information potential**. It then creates **temporaries** of that type, initializing them with the other values. This is called **promotion**.

Assignment and Mixed Expressions

What is stored in **a** in the last example? That depends on the type of **a**. If the variable is other than **double**, it is **implicitly converted** into the same type as the variable.

- **Widening** conversions occur when the assignment causes a promotion, such as from **int** to **double**. These will always succeed (as in Java and C#).
- **Narrowing** conversions occur when the assignment has the potential for loosing information, such as assigning from **double** to **int**.

Narrowing assignment conversions are prohibited in Java and C#, but **are the default in** C++. To turn off implicit narrowing conversions, C++11 added **brace** or **list assignment**; this makes C++ work more like Java and C#.

```
1  int a = 5 * 3.5;      // 17; implicit truncation
2  int b = {5 * 3.5};    // C++11; compiler error
```

Assignment and narrowing conversions.

Type Casts

Specify **explicit conversions** by using a **type cast**.

```
1  int num = 5, denom = 7;
2  double fract1 = num / denom;     // oops! now 0.0
3  double fract2 = static_cast<double>(num) / denom;
```

Using static_cast.

1. **num** and **denom** are both integers
2. **fract1** is a **double**, but the calculation uses **int**, so **fract1** is **0.0**.
3. **static_cast** creates a temporary **double** to "stand in" for **num**, so floating-point division is performed instead of integer division.

There are four of **named casts**. We'll see others later. Bjarne Stroustrup, (the inventor of C++) has several reasons why you should use these new-style casts on his C++ FAQ.

Standard Functions

A mathematical function, such as $f(x) = x^2 + 1$ means that **f(x)** computes a value equivalent to the square of **x** plus one. For any value **x**, you can compute the value of the function by applying the formula; thus **f(3)** is $3^2 + 1$, or **10**.

In C++ a **function** is a block of code that has been given a name. To run that code, you **call the function**. To call a function in C++, you write the name of the function, followed by a list of **arguments** in parentheses. We can implement *`f(2.0)`* in C++ like this:

```
double f(double x) { return x * x + 1; }
```

When called, the function copies the data supplied as arguments into the appropriate **parameter variables** (`x` in this example), and then executes the code in its body. When finished, control returns to the point in the code from which the call was made.

The operation of going back to the calling program is called **returning** from the function. A function often passes a value back to its caller. This is called **returning a value**.

The <cmath> Library

C++ has an extensive standard mathematical library called `<cmath>` that includes many of the functions you are likely to use. After including the `<cmath>` header, you can use the functions just like this:

```
double root = sqrt(value);
```

In this case, `sqrt` is the function, `value` is the argument passed into the function, and `root` is where the answer, returned from the function, will be stored. Notice, that unlike Java, we **don't** use method call syntax to call the math functions in the standard library.

Normally, you'll just look up these functions online. However, you should be able to use `sqrt()`, `abs()` and `pow()` from memory.

Finish Up

- Complete the **reading exercises** (REX) for this chapter.
- Complete **homework** using the **CS50 IDE**. Find instructions on Canvas.
  a. Make sure you submit the assignment using `make submit`.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.