Integers

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 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**. Here are the rules:

- 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. Other platforms have different limits. For instance, on the current version of Visual C++, `long` is 32 bits, just like the `int`.

Signed and Unsigned

Unlike Java, C++ integers come in two "flavors": signed and unsigned. Unsigned variables offer **twice the range of positive numbers**, 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 as `unsigned`.

Since integers use a fixed amount of memory, what happens if you exceed their range? Unsigned numbers will "wrap around". For instance, try this.

```
unsigned n = 0;
cout << n - 1 << endl;
```

As you can see, the output wraps around from zero to the largest possible `unsigned` value.

```
4294967295
```

Signed Overflow

This is not necessarily the case with signed numbers, however. Overflow and underflow on signed numbers is **undefined behavior**. Consider this code:

```
int n = 2147483647; // max size of 32-bit int
cout << "one larger is " << n + 1 << endl;
```

On many modern compilers (including ours), if you add the compiler flag `-fsanitize=undefined`, you will get a runtime error, like that shown here.

```
overflow.cpp:7:37: runtime error: signed integer overflow:
2147483647 + 1 cannot be represented in type 'int'
```

If you leave off that flag, most compilers wrap around just as with signed numbers, and it will print this:

```
one larger is -2147483648
```

Integer Literals

Explicit values like `235` or `-75` are called literals. An integer literal is a sequence of decimal digits, with no spaces or commas allowed, preceded by an optional (`+/-`) sign. It is stored as a **signed** `int`.

- Change the representation from `signed` to `unsigned` by add a `U` to the end.
- Change the storage from `int` to `long`, or to `long long` by adding an `L` or an `LL`.

Here are some examples:

```
auto a = 15;      // a is stored as a signed decimal int
auto b = 15L;     // b is stored as a signed decimal long
auto c = 15LL;    // c is stored as a signed decimal long long
auto d = 15UL;    // d is an unsigned decimal long
```

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

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
```

Starting in C++14 you can use the apostrophe as a visual separator, as I've done here to separate the digits in `bin32` into groups of 4.

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`.

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. For instance, you can write (`2.9979E+8` to represent the speed of light, instead of writing it as `299790000`.) The exponent can be positive (for large numbers) or negative (for very small numbers), 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`.

Here are some examples of floating-point literals:

```
auto a = 3.14159;       // fixed notation, type double
auto b = 2.997E8;       // scientific notation, type double
auto c = 299'792'458L;  // fixed notation, type long double
auto d = 3.5F;          // fixed notation, type float
```

Generally, use `double`, not `float` or `long double`.

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`.

Often, this is not what you want. To **explicitly** set the output format involves 3 steps, but you only need to do it 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.
3. **Specify the number of decimal places** to be displayed, using the `setprecision(n)` manipulator.

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.

- 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

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:

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

An Expression Vocabulary

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

1. 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 yeild a value
2. 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, while **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** of the expression.

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

```
int a = 3 + 7 * 5 / 2 - 4;
```

1. There are **six operands**: the variable `a`s and five `int` literals, along with **five operators**: the assignment operator, multiplication, division, addition and subtraction.
2. Multiplication and division have higher precedence than addition or subtraction. However, the `*` and `/` operators are **tied** when it comes to dealing with the `5`. That means we have to fall back on **associativity**, going left-to-right, performing the multiplication before the division.

   ```
   a = 3 + ((7 * 5) / 2) - 4;
   ```

   Using parentheses we can represent the expression at this stage like this. Evaluating those subexpressions, we end up with:

   ```
   a = 3 + ((35) / 2) - 4;   // multiplication
   a = 3 + 17 - 4;           // division
   ```

3. Now we have three operands and two operators at the same precedence. Again, we fall back on associativity (left to right) and evaluate addition (on the left) and then subtraction (on the right).

   ```
   a = (3 + 17) - 4;   // addition
   a = 20 - 4;         // subtraction
   ```

4. The assignment operator has the lowest precedence of all, so we finish up by copying `16` into the variable `a` (this is the **side effect of the expression) and returning `16` as its value.**

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:

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

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 in which the functions will be called: `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**.

Integer Division & Remainders

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**.

Integer division works like grade-school **long division**. You draw a little "house" on the board and put the "maximum occupancy" (called the **dividend**) inside the house. That is the number you want to divide.

Next, you draw the number you want to divide by (the **divisor**), standing at the front door of the house like a group of visitors. In the picture, you can see we have a dividend of `253` and a divisor of `5`.

Then you ask, "how many groups" (of 5 in this case), 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` were rounded.

The Remainder Operator

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.

Here are some examples:

```
cout << 1 / 3 << endl;        // 0 int division
cout << 1.0 / 3 << endl;      // .3333 true division
cout << 12 % 5 << endl;       // 2 left over after 12 / 5
```

Assignment Operators

With the expression `cout << 11`, the `cout` object is changed and the character pair `11` appears on the screen. Both the change to `cout` and the printing on the screen are called **side effects**. Here are some 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:

```
int x, y, z;
x = y = z = 10;       // chained assignment, which means...
x = y = (z = 10);     // right associative, which means...
x = (y = 10);
x = 10;
```

Shorthand Assignment

To **modify an existing variable**, use the **shorthand-assignment operators**:

```
x += 5;     // means x = x + 5
x -= 5;     // means x = x - 5
x *= 5;     // means x = x * 5
x /= 5;     // means x = x / 5
x %= 5;     // means x = x % 5
```

Increment and Decrement

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

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

In addition to the side effect (changing the variable), expressions using 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 the expression uses post or pre-increment.

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

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 the expression `++++a` is legal.

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

A Side-effect Pitfall

Don't ever use any side-effect operator twice on the same variable in the same expression. These expressions all result in **undefined behavior**, as you'll see if you run the code yourself in g++ , visual c++ or in clang++ .

```
int n = 6;
print(n, ++n);    // passing 6,7 or 7,7? Can't tell!
int a = n * n++;
n = n++;
cout << n++ << n++ << n++ << endl;
```

Mixed-type Expressions

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....

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

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 **temporary** values of that type, initializing them with the other values. This is called **promotion**.

Assignment and Mixed Expressions

What is stored in `a` in the example shown above? That depends on the type of `a`. If the variable is other than `double`, the value is again, **implicitly converted** into the same type as the variable. Thus, while the value calculated is `17.5`, if `a` has type `int` then only the `17` will be stored.

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

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

```
int a, b;
a = 5 * 3.5;          // 17; implicit narrowing conversion
b = {5 * 3.5};        // c++11+; compiler error
```

Type Casts

You can specify an explicit conversion by using a type cast, like this:

```cpp
int numerator = 5, denominator = 7;
double bad = numerator / denominator;   // OOPS!!! now 0
double good = static_cast<double>(numerator) / denominator;
```

1. `numerator` and `denominator` are both integers
2. `bad` is a `double`, but the calculation uses `int`, so bad ends up with `0.0`.
3. `static_cast` creates a temporary, anonymous `double` to "stand in" for `numerator` during the calculation, so floating-point (true) division is performed instead of integer division.

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

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. Here is a call to the function named ***f***, passing the argument **3**:

```
cout << f(3) << endl;
```

We can **implement** the function $f(x)$ 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>* Header

C++ has an extensive standard mathematical library **called <cmath> that** includes many of the pre-built 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** (or copied) 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, like `Math.sqrt()` to call the math functions in the standard library.

Normally, you'll just look up the math functions online online when you need them. However, you should be able to use `sqrt()`, `abs()` and `pow()` without refering to the documentation.