

# Getting Started

---

Handout written by Julie Zelenski, Mehran Sahami, and Robert Plummer.

After today's lecture, you should run home and read all of Chapters 1 and 2 on your own. We won't teach the basic syntax and constructs. We'll just highlight some of the common programming idioms and C++ idiosyncrasies.

This handout contains a smattering of facts about the C++ language, mentions some of the basics, and points out a few things to avoid.

## Most programs have the same component sections

—A comment explaining what the program does

This is a good programming practice that makes your code more accessible to the reader (who might be you!)

—Library inclusions

These **#include** statements let the compiler know what libraries you want to use functions from. Many standard libraries come with all C++ compilers. You will also use libraries you write yourself, or in this course, libraries that we provide such as **genlib** or **simpio**.

—Constant definitions

By using the **#define** mechanism to define constants, you make your program more readable and avoid "magic numbers". You also make it easy to change the value of the constant.

—Function prototypes

One common style is to write the code for functions after the **main** program. This means that functions are often called before the code appears. Function prototypes let the compiler compile the calls correctly and make sure they are consistent with the function code.

—Main program

The main program is actually a function called **main**. A prototype for this function is not required. Execution of your program begins with the first line of the main function. The function **main** takes no arguments and returns an **int** (as shown below); typically we return **0** to indicate the program exited cleanly.

—Function definitions

All programs worth writing involve breaking the code into functions, so that no part of the program manages too large a detail. Good decomposition leads to code that is clear, logical, and easy to understand. The first assignment, in fact, is an exercise in decomposition, and the A program will make it a point to write functions that can be called from two or more different places.

## Variables must be declared before they're used

Variables are local to the function in which they are declared. We will never use global variable, and you won't either. 😊 In C++, local variables can be declared anywhere with a block of statements. The scope of a local variable extends to the end of the enclosing block.

## All values have a "type", and every variable has a declared type

C++ has standard data types for storing integers, floating point numbers, Booleans, and characters (see textbook for details). The standard C++ **string** library adds **string**.

## Using cout for output

All primitive types can be inserted into a stream using the << operator. Each value will be output using the default formatting unless you change the stream state using manipulators. See the table of stream manipulators in the text. With regard to streams (for output), memorize the basics; learn the more obscure features on a "need to know" basis.

## The **simpio** functions for reading input

The **GetLine**, **GetInteger**, and **GetReal** functions are simple-to-use routines that read one piece of data from the user. These are CS106-specific functions, not standard C++, but save you from having to deal with the messy features of input until later. We rarely read from **cin**. We rely on functions provided by the **simpio** library.

## Expressions evaluate using rules of precedence and associativity

In the expression **a + b \* c**, the multiplication is performed first because of the precedence of multiplication over addition (just like you're used to in algebra), so the expression would evaluate as: **a + (b \* c)**. In the expression: **a - b - c**, the **b** is subtracted from the **a** first because of associativity. In general, put the extra parentheses in.

## Shorthand for assigning multiple variables

You can write **x = y = z = 0** in C++, causing **x**, **y**, and **z** to all be assigned the value **0**. The reason such a statement works is because an assignment is an expression that has as its result the value assigned, and the assignment operator associates to the right.

## In expressions of mixed types, values are promoted to the richer type

For example, if an **int score** is multiplied by a **double curve**, the value of the **int** is converted to type **double** before the multiplication. This does not affect the value of **score**, just the outcome of the multiplication.

## Assignment does a type conversion if necessary

Consider the following code:

```
int i;
i = 2.9;
```

This code assigns **2** to **i** since the type of the value being assigned is converted to the type of the variable that it is being assigned to. Note that floating point numbers (real values,

such as type `double`) are converted to `ints` by truncating the fractional component (e.g., `2.9` becomes `2`). Truncation and rounding are two different things.

### Integer division truncates

The expression `29/10` evaluates to `2`, since both values involved in the division are integers. If you really want a fractional result from the division, change the expression so at least one of the operands is of floating-point type—either by appending a `".0"` to an integer constant expression (`2` becomes `2.0`) or using a typecast to type `double`.

### C++ has arithmetic shorthand operators

The statements `x++` and `--x` are shorthand expressions for incrementing or decrementing the value of the variable `x` by 1, respectively. Where the `++` and `--` are placed with respect to the variable name (i.e., before or after the variable name) makes a difference in how results from that operation can be used. Importantly, be sure you understand the meaning of the following representative expressions that use some of the shorthands:

```
y = a + x++;
z = --x % a;
salary *= 2;
```

### Logical operators use "short-circuit" evaluation

The connectives representing logical "and" (denoted `&&`) and logical "or" (denoted `||`) evaluate left-to-right and stop as soon as the overall truth of the result can be determined (this latter phenomena is know as "short-circuit" evaluation). Such evaluation ordering can be taken advantage of to compute quick tests before the more lengthy ones, guard against division by zero, etc. For example, the expression below will successfully guard against trying to evaluate `x % y` if `y` has the value `0`.

```
if (y != 0 && x % y == 0)
```

### Don't confuse Boolean logic with bitwise operators

If you mistakenly use `&` (bitwise "and") where you meant `&&` (logical "and") the result might be a valid expression and compile (surprise!) but not do what you want. The single `&` and `|` are bitwise operators, the double `&&` and `||` are logical (Boolean) operators.

### Statements end with semicolons and are grouped with curly braces.

Putting a set of statements in curly braces creates a compound statement or a block. There is more than one convention for the placement of braces. Some people prefer

```
if (x < 0)
{
    y = x;
    z = 0;
}
```

Others would write

```
if (x < 0) {
    y = x;
    z = 0;
}
```

Take your pick, but be consistent within any given program! The amount of indentation is also something that is up to you, but again, should be consistently applied.

### Any non-zero expression is true

C++ has a Boolean type with values **true** and **false**, but in fact, the language considers any non-zero expression to be true, and any zero expression to be false. This means it is possible to just write:

```
if (x) {
```

Which means the same thing as:

```
if (x != 0) {
```

Pay close attention to this until it becomes second nature. Something that is wrong often compiles but produces unexpected results. Writing `=` (assignment operator) when you meant `==` (equals operator) is a common mistake.

### if and switch are conditionals

The test in an **if** statement must always be enclosed in parenthesis. A sample **if** statement is shown below:

```
if (time != 0) {
    rate = distance / time;
}
```

The **if** statement has an optional **else** part (which only evaluates if the expression in the statement is **false**). You can string **if/elses** together in a form known as a "cascading if":

```
if (rank == 1) {
    cout << "Gold medal." << endl;
    points = 10;
} else if (rank == 2) {
    cout << "Silver medal." << endl;
    points = 5;
} else if (rank == 3) {
    cout << "Bronze medal." << endl;
    points = 2;
} else {
    cout << "Consolation prize." << endl;
    points = 1;
}
```

A **switch** statement can be used to route control to different cases based on matching an integer to specific values. Re-writing the above code using **switch** would look like:

```
switch (rank) {
    case 1:
        cout << "Gold medal." << endl;
        points = 10;
        break;
    case 2:
        cout << "Silver medal." << endl;
        points = 5;
        break;
```

```

    case 3:
        cout << "Bronze medal." << endl;
        points = 3;
        break;
    default:
        cout << "Consolation prize." << endl;
        points = 1;
        break;
}

```

Be sure you understand why the **break** is there in each case and what happens if you leave it out! The use of **break** is very important in **switch** statements.

The **switch** does not allow for ranges or more complicated tests in the cases. The following code cannot be converted to a **switch**, since we are testing for ranges of values and the variable involved is of type **double**:

```

double score;
string letterGrade;

score = GetReal();
if (score > 90)
    letterGrade = "A";
else if (score > 80)
    letterGrade = "B";
else if (score > 70)
    letterGrade = "C";
else
    letterGrade = "You don't want to know.";

```

### Loop constructs available in C++ are **for** and **while** and **do-while**

The **for** loop is most commonly used to iterate forward through a sequence of numbers. Changing the initialization or increment portion of the loop allows you to create loops that count down (rather than up), loops by that increment by 2 (or some other value) after each iteration through the loop, etc.

A common **for** loop pattern is shown below:

```

sum = 0;
for (int i = 0; i < 10; i++) {
    sum += i;
}

```

We can also rewrite the **for** loop above using a **while** loop instead as follows:

```

sum = 0;
i = 0;
while (i < 10) {
    sum += i;
    i++;
}

```

In general, any **for** loop can be re-written as a **while** loop and vice versa. It is usually preferable to use a **for** loop for straightforward iterative tasks and **while** loops for those of *indefinite* (not to be confused with *infinite*) iteration.

A **do-while** loop is similar to an ordinary **while** loop except that the test is performed at the bottom of the loop rather than the top. It is rarely used, except in those situations where you are sure the loop body needs to execute at least once.

```
do {
    cout << "Your answer? ";
    response = GetInteger();
} while (response != correctAnswer)
```

### Solving the loop-and-a-half problem with **while(true)** and **break**

When trying to solve a problem of the form:

```
// get a value from user
while (value != sentinelValue) {
    // process the value
    // get a value from user
}
```

it is better to write:

```
while (true) {
    // get a value from user
    if (value == sentinel) break;
    // process the value
}
```

The idea here is that we always want to execute the first half of the **while** loop, and only stop iterating if a particular condition is met. This basic form is known as the loop-and-a-half problem. The **break** statement is used to "break out" of the loop in the middle. Note that the **break** statement can be used to exit any of the three loop types we've discussed above. If loops are nested, a **break** only exits the innermost loop. As an important consideration for good programming style, it is best to only use at most one **break** statement in any given loop (to make it clear at a single point what the condition is for exiting the loop).