C++ Syntax Basics

The `hello` program, which you saw earlier, doesn't include many features you'd expect in a "real" program. To illustrate more of C++, let's consider the program named ***`f2c`*** which converts Fahrenheit temperatures to Celsius.

This is an **IPO** or **Input Processing Output** program, typical of those that we'll be building in these first few weeks. A computer is an **information processor**; a **Cuisinart** for words, numbers and ideas. Instead of vegetables, you feed it **input**; raw numbers, facts, figures and symbols (**data**). Your computer program **processes** that data, and turns it into **information**: organized, meaningful and useful **output**.

Every program is based on this fundamental concept: input data, process it and then produce (or output) some information.

The IPO Pattern

The programs we'll be writing are console-mode, IPO programs. These are plain-text programs, which run inside your terminal, and follow this pattern:

1. **prompt** the user to enter some input
2. **read** the user's input, storing it in variables
3. **process** the input
4. **output** the results

Here's an example:

```
~/ $ ./f2c
Enter a temperature in Fahrenheit: 212
Converted: 212F ->100C
```

- The ***`f2c`*** program asks the user for a temperature in Fahrenheit. This is the **prompt**. Note that the prompt appears on the same line as the input and ends in a space, so that the input is nicely separated.
- The program stops and waits for the user to enter in some input and press **`ENTER`**. In this case, the user entered **`212`** for the temperature in Fahrenheit. The program reads the user input and stores it in a variable.
- Next, the program uses an **algorithm** to convert the Fahrenheit temperature into Celcius.
- Finally, it diplays the result on the console.

The screenshot above is called a **sample run**. It shows the input and output in the terminal or shell. Now, let's take a closer look at the ***`f2c program itself.`***

The *f2c* Program

Below you'll find the code for the *f2c* program. Use the arrow on the left to show and hide the code as we discuss its various features.

▶ *The f2c Source Code*

Comments

A **comment** is text provided for readers of your code; comments are ignored by the compiler. C++ has two kinds of comments:

- **Single-line** (`//`) are comments that end when the line ends.
- **Paired** (`/* … */`) are comments which can enclose several lines.

Documentation comments(`/** … */`) are paired comments, which start with `/**` instead of `/*`. The are processed using a tool called **Doxygen**. The *f2c* program begins with a documentation **file comment** (lines 1-6), which describes the program as a whole, and uses three **Doxygen tags**:

- `@file` contains the name of the program file and allows **Doxygen** to locate and identify the functions which it needs to document.
- `@author` should your CS 150 login id (such as **sgilbert**).
- `@version` should contain your CS 150 section (such as **Spring '24 MWAM**)

Functions also have documentation comments appearing **before** each one. Such comments start with a line describing the purpose of the functions, and then contain tags which describe the function **inputs and outputs**:

- `@param` is the name and description of each input, or **parameter**.
- `@return` describes the meaning of the output produced by the function.

Lines 24-28 in the sample code contain the documentation comment for the *convert* function. You'll learn more about documentation comments in the homework.

The Standard C++ Library

Below you'll find the code for the *f2c* program. Use the arrow on the left to show and hide the code as we discuss its various features.

▶ *The f2c Source Code*

Libraries are collections of useful, pre-written components (functions and classes). The **standard library** comes with your C++ compiler. It is divided into a number of "packages", known as **headers** in C++. Here are the headers you'll use most often. You'll meet more as the semester goes on. Find the whole list at cppreference.com.

```
#include <iostream> // input-output for reading and printing
#include <iomanip>  // formatting output
#include <cmath>    // all math functions
#include <string>   // The C++ string class
```

The `#include` directive instructs the **preprocessor** to read the declarations from the **header file** and insert them, exactly as if you had typed them into your source code.

- Instructions to the preprocessor are called **preprocessor directives**; these always appear on a line by themselves, always start with a `#`, and do **not** end in a semicolon.
- Angle brackets indicate a header is a **system library**, part of standard C++.

In *f2c*, you'll find the `#include` statements on lines 7-8.

The Standard Namespace

In C++, libraries are combined into larger groups called **namespaces**. The standard library is in the namespace called *std*, usually pronounced **standard** instead of "es-tee-dee". For CS 150, add a **using directive** to the top of your source code, like this:

```
using namespace std;
```

In *f2c*, you find this on line 9. Think of `using namespace std;` as one more incantation that the C++ compiler requires to work its magic on your code. This only works correctly inside .cpp file. In header files, you must use a different technique, which we'll cover in lecture.

The *main* Function

Below you'll find the code for the `f2c` program. Use the arrow on the left to show and hide the code as we discuss its various features.

▶ *The `f2c` Source Code*

A **function** is a **named section of code** that performs an operation. Every C++ program must contain **exactly one** function with the name `main`, which is **automatically** called when your program starts up.

- Each **statement** in the body of the `main` function is then run (or **executed**), one after another, **in order**. This concept is called **sequence**.
- When `main` has finished its work, execution of the program ends.

The `main` function contains six different statements.

1. Line 15 is an **output statement**. It prints a **prompt**, telling the user what to enter. `cout` is the standard **output stream** (similar to `System.out` in Java or `stdout` in Python). The characters in quotes (generically called a **string literal**) are sent to the screen using the **insertion operator** (`<<`).
2. Line 16 is a **variable definition** for `fahr`, a floating-point number, called `double` in C++, which is **uninitialized**.
3. Line 17 is an **input statement**. `cin` is similar to a `Scanner` object in Java, or a `file` object in Python. It reads a sequence of characters from the keyboard and stores the converted value in `fahr` using the `>>` (or **extraction**) operator.
4. Line 18 has both a **variable definition** and a function **call**. The line calls the function named `convert`, **passing** a copy of `fahr` as an **argument**. Then, it defines a variable, `celsius`, and **initializes** it with the **returned** value.
5. Lines 19 and 20 are a **single output statement**, spread over two lines. The statement combines text and variables to produce the desired result.
6. Line 21 is a `return` statement which ends the program and returns a value to the operating system. `0` indicates success, while anything else signals failure. You **may omit this `return` statement in `main`, but not in any other function.**

The *convert* Function

Below you'll find the code for the *f2c* program. Use the arrow on the left to show and hide the code as we discuss its various features.

▶ *The `f2c` Source Code*

In addition to ***main***, the ***f2c*** program **uses** the function ***convert*** which the ***main*** function **calls** (on line 18) to carry out its work. Before ***main*** can **call *convert***, though, the compiler must know what kind of arguments the function requires, and what kind of value it will return. This information is called the function's **interface**, and it is supplied by adding a **function declaration** or **prototype**, appearing **before** the ***main*** function, (on line 11).

```
double convert(double temperature);
```

The prototype provides the information needed to **call** the function: **its name along with the types of its inputs and outputs**. C++ **requires** prototype declarations so the compiler can check whether calls to functions are compatible with the definitions of those functions.

The ***convert*** function **definition**, starting on line 29, repeats the interface information from the prototype, but**does not** end in a semicolon. Instead, the definition **header** is followed by a **body** (just like ***main***) consisting of a list of statements surrounded (or **delimited**) by braces **{}**.

```
double convert(double temperature)
{
    return (temp - 32) * 5.0 / 9.0;
}
```

The body of **convert** is a single **return** statement which uses a formula or **expression** to convert the **input** Fahrenheit temperature into the **output** Celsius temperature, and return it to the caller.

Variables

Computer systems consist of hardware and software, working together to carry out the **information processing cycle**. **I** (input), **P** (processing), and **O** (output). In this section we'll to look at the objects (**variables**) that store information (**values**), and how you can create, modify and display them.

A **variable** is a **named** area in memory that stores a **value**. If you want a quick physical analogy, think of a box with a label on it. Here are the five things which you can do with a variable:

- **Declare**: associates a name with a type. (What kind of variable is it?)
- **Define**: allocate or reserve space for the variable. (Create the variable.)
- **Initialize**: provide a starting value when the variable is created.
- **Assign**: copy a new value into an existing variable.
- **Input**: read a value and use that to initialize or assign to a variable.

Declaration

Sometimes, a variable will be created in one part of your program, (or even outside of your program, by the operating system), and you want to use that variable in your code. To do so, you need to **declare** the variable. A declaration associates a variable name with a particular type, but **does not** create the variable.

```
extern string ASSIGNMENT;
```

Here is a **declaration** for a **global** variable which you can use in your homework, but **which is defined elsewhere**.

Definitions & Data Types

A definition statement allocates memory for a variable's value. You will normally combine both declaration and defintion into a **defining declaration** like this:

```
int counter;        // counter is declared an defined
char letters[10];   // letters is an array of 10 chars
string verb;        // verb is declared and defined
Star rigel;         // Star is a user-defined type
```

Each name is associated with a particular **kind of data** (ints **type**), and the compiler **allocates space** in memory to hold a value for each one. A variable may be **defined exactly once** in a program, but may be **declared** any number of times.

In your homework, you must **define** the `STUDENT` varible, the `ASSIGNMENT` is only **declared**, **not defined**. This means it must be defined elsewhere (in this case, in the procompiled **library** file `libh01.a`.)

Data Types

Variables in C++ are **strongly** typed. **Strong typing** means that each variable has a particular **data type** which does not change as the program runs. If you think of a variable as a box, you can think of a variable's data type as the **kind** of data which it can store. As an analogy, think of your local convenience store, where different product containers are specialized for a particular kind of beverage or snack.

In some languages, (such as Python), it is possible for the same variable to store a number at one point, and a string at another.

C++ is also **statically** typed. **Static typing** means that variable **types** are **explicitly specified** in your source code, unlike Python or JavaScript where they are not.

We categorize the C++ types as.

- **Built-in value types** are part of the language; also called **fundamental**, **primitive** types. In the previous section, `counter` is a built-in, primitive type
- **Derived (or compound) types** are part of the language, but are built upon one of the other types; this includes **pointers**, **arrays** and **references**. The array `letters` in example at the top of this page is a derived or compound type.
- **Library types**, such as `string` and `vector`, are class types supplied as part of the Standard C++ library; they are not part of the C++ language. In the example from the previous section, `verb` is a `string`, one of the library types.
- **User-defined types** are designed and implemented by programmers. These are **enumerated types**, **classes** and **structures**. The variable `rigel` in the example is a variable of a user-defined type named `Star`.

Identifiers & Constants

Names used for variables, functions, types, constants, classes, and so on are called **identifiers**. Here are the language **syntax rules** for identifiers:

- A name must consist of letters, digits or the underscore.
- A name must **start with** a letter or a **single** underscore, not a number.
- Names are **case-sensitive**. The name `ABC` is not the same as the name `abc`.
- A name **cannot** be one of the reserved keywords. While you **can** use names of library types (such as `string` or `vector` or `cout`) as identifiers, doing so **is just asking for trouble**; you should treat them the same as the built-in keywords.
- Only identifiers in the **standard library** may start with **two** underscores or an underscore followed by a capital (__bob and _Bob are **illegal** in user code). (The compiler can't enforce this rule, since it can't know if you are implementing part of the standard library.)

Here are the **naming conventions** we'll use in CS 150.

- Begin variables and functions with a lowercase letter: `limit` or `run()`.
- If a name consists of several words, you may use either of these:
  1. Capitalize the first letter of each word. This is known as **camelCase** and is widely used in Java.
  2. Use lowercase and underscore separators (`get_line`). This is known as **snake_case** is is common in C and Python.
- Classes and other user-defined data types should begin with an uppercase letter, as in `Alien` or `Point3D`.

Constants

Values which appear literally in a calculation are called **magic numbers**. Your code will be much more reliable and much easier to maintain, if you **replace** all magic numbers with **named constants**, similar to this:

```
const double kLocalTaxRate = .00175;
```

While you may write named constants entirely in uppercase, (`PI` or `HALF_DOLLAR`), in C++ all-caps usually indicates the presence of a preprocessor MACRO, which is discouraged. Instead, you may want to follow the Google style guide and start with a `k` (such as `kPi`).

Initialization

A variable is a named "chunk" of memory which contains a value, while a value is **a set of bits, interpreted according to its type**. Initialization provides a value **when a variable is created**.

Here are three ways to initialize a variable:

```
int a{42};    // uniform initialization
int b(35.5);  // direct initialization
int c = 4;    // legacy initialization
```

- Starting with C++11, **uniform**, **universal** or **list initialization** is the preferred way to initialize most variables. This form of initialization is value-preserving, like initialization in Java and C#. Attempts to use an initializer that would lose information (called a **narrowing conversion**), are rejected.
- **Direct initialization** uses parentheses, not braces, surrounding the initializer. Direct initialization permits **narrowing conversions**, where the initializer is implicitly **truncated** if it is too large. In the example above, the initializer `35.5` is truncated to the `int` value `35`. Direct initialization allows you to supply multiple initializers which is appropriate for many class types.
- **Legacy**) initialization is inherited from C. Like direct initialization, both widening and narrowing conversions are allowed.

What happens when variables are **not** initialized? In Java, C# and Python, **they can't be used**. (This is called the **definite assignment** rule. In C++, they **may be used**, according to these rules.

- **Primitive local** variables, which are not initialized, are **undefined**. Using such a variable is **undefined behavior** but it is not a syntax error, as in Java/C#. (Primitive variables are the built-in types like `int`, `double`, `char` and `bool`.)
- **Library variables** (such as `string`) are **automatically** initialized by implicitly calling their constructors (unlike Java).
- **Global** primitive variables are automatically initialized to `0`.

Assignment

The assignment operator copies the value on its right, and stores the copy inside the **already** existing variable on its left. Here are some examples:

```
1  int sides = 7;   // initialization (not assignment)
2  . . .
3  sides = 10;       // non-range-checked assignment
4  sides = {3.5};    // range-checked assignment (error)
```

- Line 1 is **initialization**; it **may** appear **outside** of a function.
- Lines 3 and 4 are **assignment**; they copy a value into an existing variable.
- All assignment statements **must appear inside a function**.
- **List-assignment,** with the value enclosed in braces, allows the compiler to perform additional range or type checking. Line 4 produces a compiler error because `3.5` cannot be converted to an `int` without loss of information.

Lvalues and Rvalues

An ***lvalue*** is an object that has an address. Such objects can appear on the left-hand-side of an assignment operation. (The "el" stands for "left".) An ***rvalue*** is any value which may appear on the right-hand-side of an assignment.

- A variable may be used as **either** an ***lvalue*** or an ***rvalue***.
- Literals (such as the number `10`, or the string literal `"hello"`), as well as temporaries (such as the value produced by an expression, such as `(3 + 4)`) may only be ***rvalues***.
- Constants and arrays are called **non-modifiable** ***lvalues*** since their names appear on the left side of the assignment operator when they are defined, but cannot be modified later.

Console Output

C++ uses an object-oriented library named `<iostream>` for input and output. The C++ standard library contains several predefined **stream objects**. Here are two:

- **`cout`**: **standard output**; similar to **`System.out`** in Java or **`stdout`** in Python.
- **`cin`**: **standard input**; similar to a **`Scanner`** object in Java or **`stdin`** in Python.

To use these objects, include these headers:

```
#include <iostream>   // standard stream objects
#include <iomanip>    // "manipulators" for output formatting
```

The **manipulators** in **`<iomanip>`** control the formatting of real numbers.

Streams can be thought of as **data flowing sequentially from** a source that produces it, and **flowing to** a destination, where it is displayed or saved. You **insert** a value into the stream and it eventually reaches its destination.

The **insertion** (or output) **operator** is the symbol pair (**`<<`**) pointing **to** an **output stream object.** On the right of the operator are the values to insert into the stream.

```
cout << "I am now " << 73 << " years old!" << endl;
```

- The words **`"I am now"`** are called a **string literal**, text enclosed in double quotes.
- Numbers **are not enclosed in quotes**; **`cout`** has the ability to convert binary values into their textual form.
- To end output line, you can use the **newline** escape character (**`\n`**) or the **`endl`** (*end-el*) **stream manipulator** object as is done here.

An output statement may **insert several values** into the stream, but each must have its own insertion operator.

If you need to print special characters (like a double quote, or a backslash), then use the same sort of **escape sequences** that you employed in Java or Python:

```
cout << "\"Hooray\", the crowd cheered!" << endl;
```

Console Input

The `cin` (*see-in*) standard input stream can read and convert user input, and store it into different kinds of variables. This is called **formatted input**, and it uses the **extraction operator** (`>>`) to read (extract) data from input, **convert it and store** the results in a variable.

Here's an example:

```
1  cout << "Enter limit: ";  // prompt
2  int limit;                // variable to hold the value
3  cin >> limit;             // read, convert and store
```

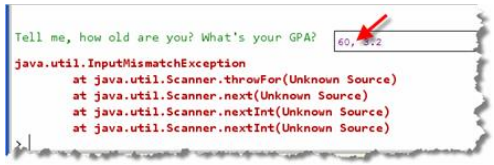
When a user types `123` in response when prompted for `limit`, the input is three `ASCII` character values `'1'`, `'2'`, `'3'`. These are stored sequentially in memory, and then, when the user types `ENTER`, the three `char` values are **combined and converted** from text into to the `int` `123`, which looks like this in memory:

```
0000-0000 0000-0000 0000-0000 0111-1011
```

This process—turning human-readable text into binary numbers, (and it's the reverse), is the job of **parsing** or **conversion**. **The `cin` object does this for us.**

Input Errors

If the user **types an unexpected input value** in Java or C# or Python, the system **prints an error message on the console**, and terminates the program.



This is a **runtime error** or **exception**, detected when your program runs, rather than when you compile it. **C++ uses a different technique**.

Instead of causing a runtime error and stopping, the input stream is **placed in an invalid state**, and stops receiving input. In C++ when the comma is encountered, your program **doesn't crash**. You'll learn how to handle these kinds of errors soon.