Chapter

2

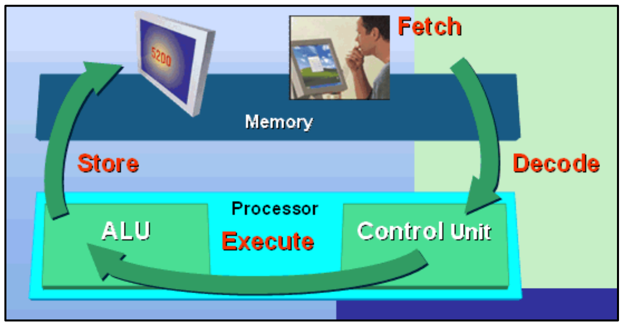
The C++ Backstory

In the 1930s, **Alan Turing** (dramatized in the Imitation Game), imagined a **universal computing machine** which could store both its data, **and** its **programs** in memory. 1940s-era computers, such as the ENIAC, **had no programs** as such. Instead, they were **hard-wired** to perform specific calculations. Programming them entailed rewiring their circuit cabinets, a process that could take several days.

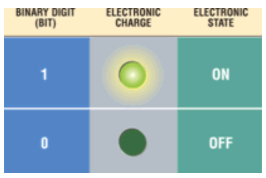
Turing's ideas were realized in 1946, when mathematician and physicist **John Von Neumann** described the first real **stored program** computer system, the EDVAC, whose **machine language** instructions were stored in memory as a **binary numeric code**.

```
1000 1011 0100 1110 0000 0110
```

To run a program on the **EVAC**, each instruction was **fetched** from memory by the CPU Control Unit (CU) and then stored in **registers** on the CPU. The instruction was then **decoded** and **executed** by the CPU's Arithmetic/Logic unit (ALU). Finally, the results were written back into memory where they could be examined. This sequence-read, decode, execute and store-is known as the instruction cycle; it's how a computer works.



Here's an example. The **Intel** CPU instruction which **copies a value from memory** into the `CX` register is `8B4E06`. We humans see this as a **hexadecimal (base 16) number**. The computer, on the other hand, is a **digital electronic device**, which **doesn't know about numbers at all**; it is built using integrated circuits (or **transistorized switches**), each of which is either on or off. **Humans interpret** the "on" state as a `1` and the off state as a `0`.



BINARY DIGIT (BIT)	ELECTRONIC CHARGE	ELECTRONIC STATE
1	🟡	ON
0	●	OFF

So, how does the computer "know" what to do with the instruction `8B4E06`? It doesn't! In **binary** this instruction is `100010110100111000000110`, but inside the hardware, it is a block of switches (or transistors). **Electricity flows** through each **1** to another part of the device. The flow of electricity is blocked by a **0**.

As a physical analogy, imagine the player-piano roll, where a hole in the paper causes a note to be played, allowing a hammer to strike a particular string.

Machine code is also called **native code**, since the computer can use it without any translation. Machine language programs are difficult to understand and, inherently **non-portable**, since they are designed for a single type of CPU.

Yet, high-performance programs are still written in machine language (or its symbolic form, assembly language). You can examine the native code for the APPLE II Disk Operating System, written by Steve Wozniak, at the Computer History Museum.

High-Level Languages

John Backus

Machine and assembly language are **low-level languages**, tied to a specific CPU's instruction set. In the mid-1950s, **John Backus** lead a team at IBM which developed the first high-level programming **language.** FORTRAN (or the **FOR**mula **TRAN**slator), allowed scientist and engineers to write their own programs.

COBOL (the **CO**mmon **B**usiness **O**riented **L**anguage), allowed accountants and bankers to write programs using a vocabulary with which they were comfortable. In 1958, **John McCarthy** at MIT built a third high-level language named LISP (the **LI**st **P**rocessing **L**anguage) to help him with his research into artificial intelligence.

These three high-level languages allowed non-computer specialists to write their own programs, but they were not **general-purpose** languages; scientists could not use COBOL to write code for NASA, and accountants could not use FORTRAN.

In 1960, the designers of **FORTRAN**, **COBOL** and **LISP** gathered together in Paris to remedy that, producing the **Algo**rithmic **L**anguage (ALGOL). Modern languages derive much of their syntax and many core concepts from ALGOL.

This photo, taken at the 1974 ACM Conference on the History of Programming Languages, shows six of the original participants who attended the 1960 Algol Conference in Paris. Top row: John McCarthy (LISP), Fritz Bauer, Joe Wegstein (COBOL). Bottom row: John Backus (FORTRAN), Peter Naur, Alan Perlis.

New languages followed rapidly in the next seven decades. Here are two:

- BASIC, the **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode, was developed at Dartmouth University in 1964. The professors **Kemeny** and **Kurtz** wanted to teach programming to university students using the new, interactive, time-share computers. **BASIC** was later the first language available on micro-computers, implemented by Bill Gates for the Altair. Even later, in the 1990s, Microsoft introduced **Visual Basic**, a graphical version, popular in the business world.

- In 1972, the Swiss computer scientist Nicholas Wirth felt that **BASIC** was teaching students bad programming habits, so he created the popular teaching language Pascal, (named after the philosopher, Blaise Pascal). Strongly influenced by **Algol**, **Pascal** enforced structured programming techniques. It was the first programming language taught in most university computing programs until about 2000. Wirth later designed **Modula**, **Oberon**, and Ada, a language still in wide use in avionics and in defense.

The C & SIMULA Languages

In 1970, two researchers at Bell Labs in New Jersey (**Ken Thompson** and **Dennis Ritchie**) developed a **portable operating system** for the new **mini-computers** just coming on the market. They named their operating system Unix (or UNIX if you like)[1]

[1] Unlike the other technologies we have discussed, UNIX is not acronymic (like FORTRAN). The original 1974 CACM paper—*The UNIX Time-Sharing System*—used all caps because, in Ritchie's words, "*we had a new typesetter and troff had just been invented and we were intoxicated by being able to produce small caps*." Later Ritchie tried to change the spelling to 'Unix' in a few of his papers. He failed and eventually gave up. See the Jargon file.

`Unix` was originally written in assembly language. Later was converted, (or ported), to a high-level language named **`B`**. Then, to simplify the development of Unix, Dennis Ritchie modified the **`B`** language and created the programming language named C.

Both **`Unix`** and **`C`** have had enormous impacts on the field of computing. Many of today's most important language are based on C, including C++, Java and C#, which still use much of the original syntax designed by Ritchie.

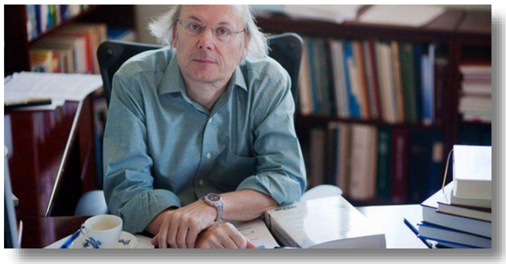
Object-Oriented Programming

`C` and **`Pascal`** are **imperative** languages, where the emphasis is on the **actions** that the computer should do. In the **procedural paradigm**, (or programming style), programs consist of a hierarchy of subprograms (**procedures** or **functions**) which process external data.

Object-oriented programming, or **`OOP`**, is a different style of programming. In the **`OOP`** paradigm, programs are communities of self-contained components (called **objects**) in which **data** and **actions** are combined.

The first object-oriented language was **`SIMULA`**, a **simulation** language designed by the Scandinavian computer scientists **Ole-Johan Dahl** and **Kristen Nygaard** in 1967. Much of the terminology we use today to describe object-oriented languages comes from their original 1967 report.

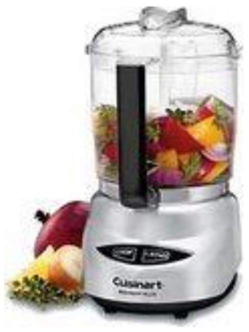
The C++ Language



One Danish Ph.D. student at Cambridge University, **Bjarne Stroustrup**, was heavily influenced by SIMULA. In 1980, working as a researcher at AT&T Bell Labs, he began adding object-oriented features to C. This first version of what would later become C++, was named C with Classes.

Unlike **idealistic** languages that enforce "one true path" to programming Utopia, C++ is **pragmatic**. C++ is designed as a multi-paradigm **language**. In C++ the object-oriented and the procedural programming styles are **complementary**, since you may find a use each approach. So, consider each paradigm or style a **tool** that you can pull out of your toolbox when needed, to apply the conceptual model that is most appropriate for the task at hand.

C++ Syntax Basics

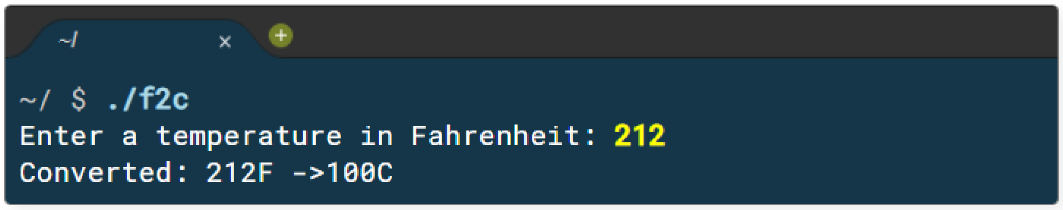


The *hello* program from the last chapter doesn't include many features you'd expect in a "real" program. To illustrate more of C++, let's consider the program named *f2c* which converts Fahrenheit temperatures to Celsius.

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.

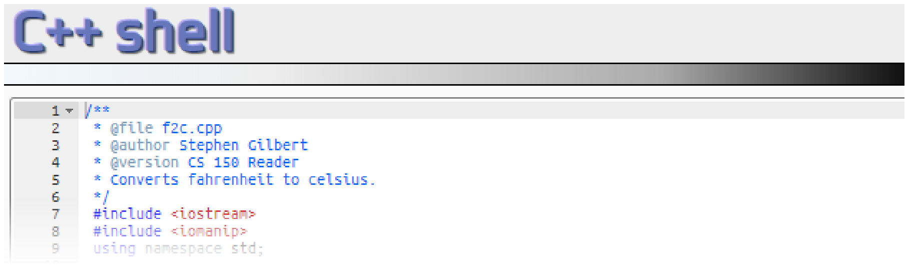
The *f2c* Program



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

The *f2c* program asks the user for a temperature in Fahrenheit, converts it and displays the result on the console. The screenshot above is called a **sample run**. It shows the input and output in the terminal or shell.

Click on the "running man" to open a copy of `f2c.cpp` in the online IDE **cpp.sh**. You'll find a link to this IDE on the Canvas home page. This is a different IDE that is useful for trying out (and sharing) ideas. It is often faster than using the Sandbox or the CS50 IDE.



```
/**
 * @file f2c.cpp
 * @author Stephen Gilbert
 * @version CS 150 Reader
 * Converts fahrenheit to celsius.
 */
#include <iostream>
#include <iomanip>
using namespace std;
```

Keep the program open in another tab for the next few pages as we talk about it.

Program Comments

A comment is text provided by readers of your code; they 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 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.
- `@author` contains your CS 150 login id (such as **sgilbert**).
- `@version` contains your CS 150 section (such as **Spring '20 MWAM**)

Functions also have documentation comments appearing before each one. They start with a line describing the purpose and then 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 contain the documentation comment for the ***convert*** function. You'll learn more about documentation comments in the homework.

The Standard C++ Library

Libraries are collections of useful, pre-written components. The **standard library** comes with your C++ compiler. It is divided into a number of "packages", known as **headers**. 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>    // c++ style strings
```

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

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:

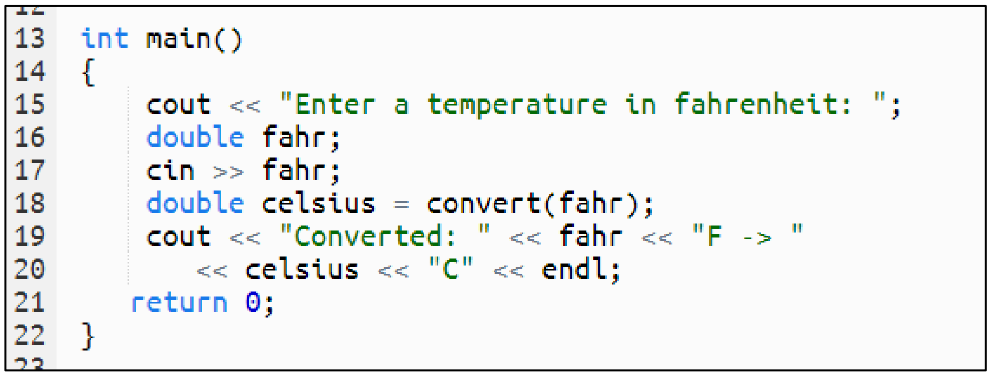
```
using namespace std;
```

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.

The main function

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 and returns, execution of the program ends.



```
13  int main()
14  {
15      cout << "Enter a temperature in fahrenheit: ";
16      double fahr;
17      cin >> fahr;
18      double celsius = convert(fahr);
19      cout << "Converted: " << fahr << "F -> "
20          << celsius << "C" << endl;
21      return 0;
22  }
```

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). The characters in quotes (a **string literal**) are sent to the screen using the insertion operator (`<<`).
2. Line 16 is a variable definition for `fahr`, a `double`, which is uninitialized.
3. Line 17 is an **input statement**. `cin` is similar to a `Scanner` object in Java. 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** statement. The line calls the function **`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, anything else signals failure. You may omit this **`return`** statement in **`main`**, but not in other functions.

The *convert* Function

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.

```
double convert(double temp);
```

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

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

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

The body of **`convert`** is a single **`return`** statement that uses a formula or **expression** to convert the **input** Fahrenheit temperature into the **output** Celsius temperature.

Variables & Values

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.

Variables

A **variable** is a named storage area in memory that holds a **value**. There are five things you can do with a variable:

- Declare: associate the name with a type (what kind is it?)
- Define: allocate space for 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 a variable.

Declaration

A declaration associates a name with a particular type, but does not create the variable. Here is a declaration for a **global** variable which is defined elsewhere.

```
extern string ASSIGNMENT;
```

Declaration for a variable.

Definition

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

```
int counter;        // counter is declared and 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** (its 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, the `STUDENT` variable is defined, but `ASSIGNMENT` is only declared (not defined), which means it must be defined elsewhere (in the precompiled library file `libh01.a`.)

Data Types

C++ is **strongly** typed, and **statically** typed. **Strong typing** means that each value has a particular **data type**. **Static typing** means that variable types are explicitly specified. We categorize the C++ types as.

- Built-in value types are part of the language; **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; include **pointers**, **arrays** and **references**. The array `letters` in the previous example 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 previous example, 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**.

Identifiers

Names used for variables, functions, types, constants, etc. are called **identifiers**.

- The name must **start with** a letter.
- All other characters in the name must be letters, digits, or the underscore.
- Names are **case-sensitive**. The name **ABC** is not the same as the name **abc**.
- The name **cannot** be one of the reserved keywords. You **can** use names of library types (such as `string` or `vector`) as identifiers, but doing so is just asking for trouble; treat them the same as the built-in keywords.

Conventions

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 English words, **you** may:
  - ➝ Capitalize the first letter of each word (`camelCase`)
  - ➝ Use lowercase and underscore separators (`get_line`)
- Classes and user-defined data types should begin with an uppercase letter, as in `Alien` or `Point3D`.
- Write named constants entirely in uppercase, (`PI` or `HALF_DOLLAR`) or follow the Google style guide and start with a `k` (such as `kPi`). In C++ all-caps usually indicates the presence of a preprocessor `MACRO`, which is discouraged.

Constants

Values that 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;
```

Values

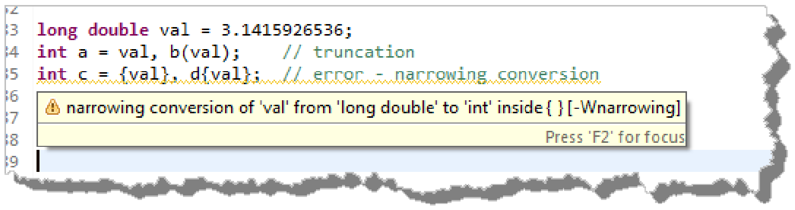
A variable is a named "chunk" of memory that contains a **value**. A value is **a set of bits, interpreted according to its type**. We store a value in a variable in three different ways.

Initialization

Initialization provides a value **when a variable is created**.

```
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.



```
long double val = 3.1415926536;
int a = val, b(val);    // truncation
int c = {val}, d{val};  // error - narrowing conversion
```

⚠ narrowing conversion of 'val' from 'long double' to 'int' inside { } [-Wnarrowing]

Press 'F2' for focus

- **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 when appropriate.
- **Assignment (or legacy)** initialization is inherited from C. Like direct initialization, both widening and narrowing conversions are allowed.

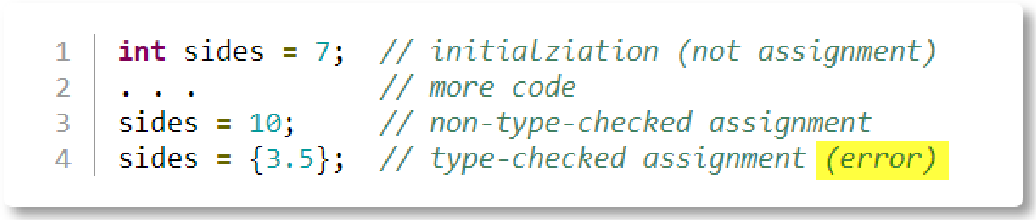
What happens when variables are **not** initialized? In Java and C#, they can't be used. In C++, they may be used, according to these rules.

- **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 use 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;   // initialziation (not assignment)
2  . . .            // more code
3  sides = 10;      // non-type-checked assignment
4  sides = {3.5};   // type-checked assignment (error)
```

Initialization and assignment.

- The first statement is **initialization**; it may appear **outside** of a function.
- Lines 3 and 4 are **assignment**; they copy a value into an existing variable.
- Assignment statements **must appear inside a function**.
- **List-assignment**, with the value enclosed in braces, allows the compiler to perform additional 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, and can appear on the left-hand-side of an assignment operation. (The "el" stands for "left".) An ***rvalue*** is a value which may appear on the right-hand-side of an assignment. Variables may be used as **either *lvalues*** or ***rvalues***. **Literals** and **temporaries** may only be ***rvalues***.

Console Input and 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.
- `cin`: standard input; similar to a `Scanner` object in Java.

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.

Console Output

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 " << 72 << " years old!" << endl;
```

The words "I am now" is 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. Finally, 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:

```
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. Called **formatted input**, it uses the **extraction operator** (`>>`) to read (extract) data from input, **convert it and store** the results in a variable.

Here's an example:

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

Prompt and read input.

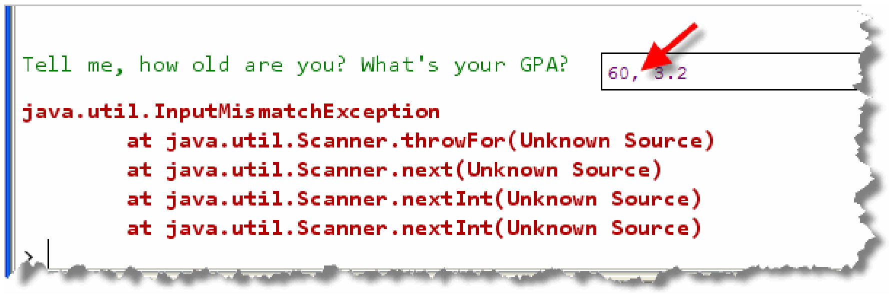
When a user types `123` in response when prompted for `limit`, the input is three `ASCII` character values `'1'`, `'2'`, `'3'`. These are stored sequentially in memory, and then, the three `char` values are **combined and converted** from text into to the **integer** `123`:

```
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.



```
Tell me, how old are you? What's your GPA?   60, 3.2
java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
>
```

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. We'll learn how to handle that soon.

Finish Up

- Complete the reading exercises **(REX)** for this chapter.
- Complete the 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.