Annotated First C++ Programs

C++ is a superset of C. It supports a variety of programming techniques: procedural programming, data abstraction, object-oriented programming, and generic programming. Most programming solutions to nontrivial problems combine aspects of these techniques. Procedural programming is concerned with structuring a program as a collection of functions with a *main* function calling several other functions, which in turn call others. Procedural programming is what C was designed to support. This document assumes you're familiar with the material from High-Level Programming 1, that is, you can read and write programs in the C programming language.

Things to know before writing first C++ program:

- By default, GCC C++ compiler `g++` and Clang C++ compiler `clang++` expect source files to have `.cpp` suffix.
- High-Level Programming 2 will use the ISO C++17 standard. Code presented in this document and throughout this semester may not compile in older C++ standards.

The minimal C++ program

Consider the following source file `minimal.cpp` that does nothing:

```
1  // this is the minimal C++ program
2  int main()
3  {
4  }
```

C++ has the exact same conventions for comments as C11. Single-line comments are specified by the double slash token `//`. A multi-line comment is program text delimited by tokens `/*` and `*/`.

Just as with a C program, every C++ program must have exactly one global function named `main`. The program starts by executing that function.

Just as with C, curly braces are used in C++ to group together stuff. The curly braces on lines 3 and 4 in `minimal.cpp` enclose an empty function body.

To `return` or not from `main` function

The `int main()` says that we're defining a function named `main` that returns a value of type `int`. The `int` value returned by `main` is the program's return value to the system. A zero value indicates success; any other value means there was a problem. Notice the definition of `main` doesn't contain an explicit `return` statement since C++ standards specify that if control reaches the end of `main` without encountering a `return` statement, the effect is that of executing

```
return 0;
```

> ***Whether to add an explicit `return` statement in `main`'s' definition or to leave it for the implementation to insert one is a choice made by individual programmers.***

`void` parameter lists

The empty parentheses in `main()` indicates that the parameter list for `main` is empty and that `main` receives nothing or no inputs from the implementation. There is nothing that prevents you from adding keyword `void` in the parentheses, as in `main(void)` to indicate that function `main` receives no parameters. However, the convention followed by C++ programmers is to leave the parentheses empty.

The *incorrect* practice of writing `int main()` in a C program has a completely different meaning. In C, a function declared as

```
int foo();
```

indicates that function `foo` can take *any number and type* of parameters. The earliest versions of C didn't require the programmer to explicitly declare the number and type of parameters in the parameter list. This turned out to be a disaster and later versions of C "encouraged" function prototypes so the compiler could check for bad behavior. As consistently practiced in HLP1, if a C function truly does not take any arguments, then the `void` keyword is used to indicate this fact:

```
int foo(void);
```

In C++, the two functions have the same declaration. The lack of parameters means that the function does not accept any arguments. It's a stylistic issue as to whether or not the `void` keyword is used.

> ***In C++, you can specify that a function takes no parameters by using an empty parameter list or by using keyword `void` in the parameter list.***

Functions must be declared before their first use

Consider a C source file `add.c` defining a function `add`:

```
double add(double a, double b) {
  return a+b;
}
```

Next, consider a C source file `main.c` that calls `add` function defined in source file `add.c`:

```
#include <stdio.h>

int main(void) {
  printf("%f\n", add(10.0, 20.0));
  return 0;
}
```

The C11 compiler will compile `main.c` but issue a diagnostic message indicating that since there is no declaration of function `add`, it will assume that function `add` will take an unknown number of parameter of unknown types and return an `int` value. The C11 compiler doesn't issue an error message because it has to let code written before C11 starting from the invention of C to be compiled. The C11 compiler will also issue a second warning indicating that format specifier `f` specifies a `double` value but `add` is returning an `int`. To ensure that such warnings were converted to errors, HLP1 required options `-pedantic-errors`, `-Wstrict-prototypes`, `-Wall`, `-Wextra`, `-Werror` when using `gcc` or `clang` compilers with C11 code.

Things are better with C++. C++ requires all functions be declared before their first use and therefore the `-Wstrict-prototypes` option is no longer required.

> ***In C++, every function, in fact every name, must be declared before its first use.***

`g++` and `clang++` compiler drivers

Open a Windows command prompt. Change your current working directory to `C:\sandbox`. Switch to Linux bash shell using Window shell command `wsl`. Open Code from the bash shell using command: `code minimal.cpp`. Use Code to enter the code described in `minimal.cpp`. After saving the file, use GCC C++ compiler `g++` to *compile* `minimal.cpp` (note that the `$` symbol below represents the Linux bash shell command prompt and is not part of the `g++` command):

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c minimal.cpp -o minimal.o
```

The five options to `g++`: `-std=c++17`, `-pedantic-errors`, `-Wall`, `-Wextra`, `-Werror` are required and necessary every time you compile a source file. More information on these options can be obtained here.

- Option `-std=c++17` means we want to use ISO C++17 standard.
- Option `-pedantic-errors` means strict compliance with ISO C++ and GCC must issue an error message when non-compliance is encountered in source code.
- Option `-Wall` will enable all warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning).
- Option `-Wextra` will enable some extra warning flags that are not enabled by `-Wall`.
- Option `-Werror` will convert all diagnostic warning messages into errors preventing successful compilation until the source code is modified to prevent any diagnostic messages to be issued by `-Wall` and `-Wextra`.
- Option `-c` indicates that source file `minimal.cpp` is only to be compiled but not linked. That is, source file `minimal.cpp` is converted to an *object file* containing binary machine code for a specific CPU but not into an executable program.
- Option `-o` gives the name `minimal.o` to the output object file generated by the compiler. All though compilers will default the object file's name to the name of the source file with an extension of `.o`, I explicitly specify the name of the object file.

Link object file `minimal.o` with standard library object files to create an executable:

```
$ g++ minimal.o -o minimal.out
```

Since this particular example describes a minimal C++ program, there are no C++ standard library object files to be linked to the object file `minimal.o`. If option `-o` is not used, the linker will default the executable file to `a.out`.

To run executable program `minimal.out`, type the executable's pathname like this:

```
$ ./minimal.out
```

Things to do

1. Repeat the compile and link steps using *verbose* option `-v`. This option prints the commands that execute the different compilation stages to standard output.
2. C++ is a complex language and new features are being continuously introduced. Therefore, a second popular compiler such as Clang is necessary to verify that new features are properly understood by the program. It is easy to compile with the Clang C++ compiler `clang++` - it uses exactly the same options as `g++`. Installing Clang is straightforward from your bash shell:

```
$ sudo apt-get update # update Linux package index
$ sudo apt-get upgrade # upgrade old packages
$ sudo apt-get install clang-10 # install Clang package
```

Second C++ program: Writing to standard output stream

Consider the code in source file `greeting.cpp` that prints a greeting to standard output:

```
 1  #include <iostream>
 2  
 3  // print a greeting to the world!!!
 4  int main() {
 5    std::cout << "The Answer to the Great Question... Of Life, the Universe\n"
 6              << "and Everything... Is... "
 7              << 6*7
 8              << ", ' said Deep Thought,\nwith infinite majesty and calm.\n"
 9              << "- Douglas Adams, The Hitchhiker's Guide to the Galaxy."
10              << std::endl;
11  }
```

Header file

Input and output are not part of the core language but are provided by the standard C++ library. They must be included explicitly; otherwise, we cannot read or write. We request input-output functionality using the `include` preprocessor directive on line 1:

```
1  #include <iostream>
```

- The name `iostream` suggests support for sequential, or stream I/O, rather than random-access or graphical I/O. Because the name `iostream` is enclosed in angle brackets (`<` and `>`), it refers to a part of the C++ library called a standard header. The preprocessor will begin searching for file `iostream` in standard include paths that were established when the compiler was installed on a computer.
- Unlike with headers in the C standard library, we don't add a `.h` suffix to the standard header file name.

Namespaces and scope operator `::`

Recall that C requires every type, function, or variable defined at the global scope to have a unique name. That is, a global variable `foo` defined in one source file and a function `foo` defined in a second source file cannot coexist in the same executable. In many cases, programmers declare types and objects that are only referenced by functions in the same source file. C provides storage specifier `static` to make the name of a function or global variable private to the source file where they're defined. This is one possible way to prevent the linker from having two different objects referenced by the same name `foo`. Avoiding name clashes in large scale programming is difficult because it is impossible to predict the names of thousands of variables, functions, and type introduced by the development team and library vendors. C++ introduces the mechanism of *namespaces* for expressing that some declarations belong together and that their names shouldn't clash with other names.

In expression `std::cout` on line 5, `std` refers to the namespace organizing together names of types, functions, and variables defined by the standard C++ library; `cout` (explained below) refers to the name of an object defined by the standard C++ library; and `::` is the *scope resolution operator* that is used when we want to refer to the name `cout` defined in namespace `std`.

Detailed coverage of namespaces in class lectures detail is delayed to a later date. See Sections $2.2.4$ for a review of scope; Section 3.1 for brief introduction to namespace `using` declaration; and Section 18.2 for a detailed overview of namespaces.

Output stream class, object, and operator `<<`

Following the principles of object orientation and generic programming, the C++ standard library models a stream as an object with properties defined by a class. A C++ *class* is a data type, analogous to `int`s, `double`s and structures, that consists of a collection of data and the functions necessary to control and maintain that data. A C++ *object* is a specific variable having a class as its data type. The most important stream classes are class `std::istream` to define input streams that can be used to read data and class `std::ostream` to define output streams that can be used to write data. Both these classes can be declared by including header file `<iostream>`. This header file defines several global (recall that global objects have external linkage and static duration) objects of type `std::istream` and `std::ostream`:

Stream Object	Stream Type	Corresponding C stream	I/O Device	Buffered
`std::cin`	input	`stdin`	keyboard	yes
`std::cout`	output	`stdout`	monitor	yes
`std::cerr`	error	`stderr`	monitor	no
`std::clog`	logging	No C equivalent	monitor	yes

Object `std::cin` of type `std::istream` and object `std::cout` of type `std::ostream` are created when the system starts your program with each object providing a stream to allow you to communicate with your console - the screen and keyboard setup that makes up the traditional computer user interface. Input stream `std::cin` (console input) connects the keyboard to your program and provides a way for reading user input. Output stream `std::cout` (console output) connect your program to the display monitor for displaying regular program output.

Object `std::cerr` is also of type `std::ostream` and provides a stream for writing error messages during program execution to the display monitor. With separate streams to write regular program output and error messages during program execution, programmers can avoid mixing program output and error messages by redirecting program data to a file while error messages are streamed to the monitor.

C++ allows special meanings to be given to operators when they're used with user-defined classes - this is referred to as *operator overloading*. Class `std::ostream` overloads the bitwise left-shift operator `<<` to *insert* characters and character strings into an output stream. In addition, the class also provides a variety of member function overloads of operator `<<` for arithmetic types, stream buffers, and manipulators. Since left-shift operator `<<` is overloaded to insert characters into an output stream, it is also known as an *inserter* or *stream insertion operator* or more casually *output operator*. Thus, the statement

```cpp
std::cout << 'h';
```

prints character h to the screen, while

```cpp
std::cout << 15;
```

prints the character representing integer value 15 to the screen. The statement

```cpp
std::cout << 3.14;
```

prints the characters representing floating-point value 3.14 to the screen. The statement

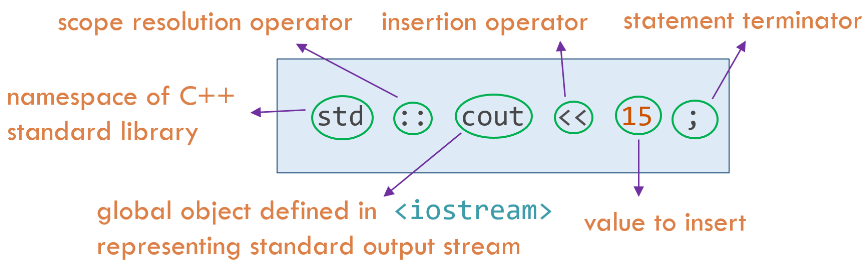
```cpp
std::cout << "hello";
```

prints individual characters in null-terminated string literal `"hello"` to the screen.

Breaking down the statement

```cpp
std::cout << 15;
```

into tokens, we have:


scope resolution operator
insertion operator
statement terminator
namespace of C++ standard library
std
::
cout
<<
15
;
global object defined in <iostream> representing standard output stream
value to insert

The overloaded `<<` functions return a reference to stream object `std::cout` and therefore it is possible to chain together multiple calls to insertion operator `<<` into a single statement:

```cpp
std::cout << 'h' << 15 << 3.14 << "hello";
```

This chaining together of multiple calls is possible because the left-shift operator is left associative. The chained calls are evaluated as:

```
((((std::cout << 'h') << 15) << 3.14) << "hello");
```

with expression `std::cout << 'h'` evaluating to `std::cout` with the side effect of writing character `'h'` to standard output stream. Next, the resulting expression `std::cout << 15` is evaluated followed by `std::cout << 3.14` and finally followed by `std::cout << "hello"`.

This should provide you with a basic understanding of the code in lines 5 through 9 in source file [`greeting.cpp`](). The code on line 10:

```
<< std::endl;
```

contains a manipulator that terminates writes to standard output stream. Manipulators are special objects that, not surprisingly, manipulate an output stream by changing the way output characters are formatted or manipulate an input stream by changing how input characters are interpreted. Manipulator `std::endl` defined in `<iostream>` means "end line" and does two things:

- outputs a newline (that is, character `'\n'`)
- flushes output stream's buffer by forcing a write of all buffered data for the given stream.

The output of the program is:

```
The Answer to the Great Question... Of Life, the Universe
and Everything... Is... 42, ' said Deep Thought,
with infinite majesty and calm.
- Douglas Adams, The Hitchhiker's Guide to the Galaxy.

```

Third C++ program: List initialization

If an initializer is specified for an object, that initializer determines the initial value of an object. An initializer can use one of four syntactic styles:

```
int i1 {5}; // list initialization
int i2 = {5};
int i3 = 5;
int i4(5);
```

Of these, only the first can be used in every context, and is strongly recommended. It is clearer and less error-prone than the alternatives. However, the first form is new in C++11, so the other three forms are what you find in older code. The two forms using `=` are what you use in C. It is generally considered ok to use C syntax for simple types:

```
int i1 = 0;
int ch = 'z';
```

However, anything more complicated than that is better done using list initialization (braces).

Fourth C++ program: Reading from standard output stream

Input stream class, object, and operator `>>`

Object `std::cin` (console input) of type `std::istream` is created when the system starts your program. It represents the standard input stream that connects the keyboard to your program and provides a way for reading user input. Since object `std::cin` is defined in `<iostream>`, you must include this file to write to the standard input stream.

Class `std::istream` overloads the bitwise right-shift operator `>>` to *extract* characters and character strings from an input stream. In addition, the class also provides a variety of member function overloads of operator `>>` for arithmetic types, stream buffers, and manipulators. Since operator `>>` is overloaded to extract characters from an input stream, it is also known as an *extractor* or *stream extraction operator* or more casually *input operator*.

Consider the following code in source `input.cpp`:

```cpp
#include <iostream>

int main() {
  std::cout << "Enter an int value: ";
  int i;
  std::cin >> i;
  std::cout << "Enter a float value: ";
  float f;
  std::cin >> f;
  std::cout << "Enter a double value: ";
  double d;
  std::cin >> d;
  std::cout << "Enter a character string: ";
  char s[10];
  std::cin >> s;

  // display values entered by user
  std::cout << i << '\n';
  std::cout << f << '\n';
  std::cout << d << '\n';
  std::cout << s << '\n';
}
```

Suppose source file `input.cpp` is compiled and linked to executable `input.out`. When this program runs, it will prompt you:

```
Enter an int value:
```

The cursor will stay on the same line as the prompt, and you should enter an integer number, say 45 followed by the Enter key. At the second, third, and fourth prompts, enter the values 3.14, 3.18888889, and string digipen. The entire interaction with the program will look like this:

```
$ ./input.out
Enter an int value: 45
Enter a float value: 3.14
Enter a double value: 3.18888889
Enter a character string: digipen
45
3.14
3.18889
digipen
$
```

Reading an integer number into variable `i` is achieved by the statement:

```
cin >> i;
```

When this statement is executed, the program waits for the user to type in a number and press the Enter key. The number is then placed into the variable, and the program executes the next statement(s). Note that none of the variables were initialized because the input statements move values into these variables. The variables are defined as close as possible to their first use - the input statement indicating where the value is set. When an integer is to be read from the keyboard, zero and negative numbers are allowed as inputs but floating-point numbers are not. If the user nevertheless provides a floating-point input, only the integer part is read.

Because operator `>>` in class `std::istream` is overloaded, floating-point values and strings can also be read. If the user inputs an integer value for floating-point variables, the value stored in the floating-point variable will have zero fractional value. When reading characters to store into a character array, `std::cin` will behave like `std::scanf` by null-terminating `char` array `s` after reading the characters comprising the string `"digipen"`. Also, note that by default, `std::cin` writes exactly 6 digits for the double-precision value of type `double`.

Setting field width to restrict number of characters read by `std::cin`

A second sample run of the code fragment with the executable from `g++` for values 10, 23.45, $.8798$, and string `"Supercalifragilistic"` reports an error:

```
Enter an int value: 10
Enter a float value: 23.45
Enter a double value: .898
Enter a character string: Supercalifragilistic
*** stack smashing detected ***: terminated
Aborted
```

while running the executable from `clang++` for the same values produces incorrect output:

```
Enter an int value: 10
Enter a float value: 23.45
Enter a double value: .898
Enter a character string: Supercalifragilistic
10
8.02427
8.89493e+252
Supercalifragilistic
```

The problem arises because of a buffer overflow caused by writing more than 9 characters in array `s` (that is defined with size 10). Researching the functions presented by the interface for type `std::istream`, a function `ios_base::width` can be used to ensure that no more 9 characters are read and stored in array `s` (so that the 10th character can be `'\0'` to ensure `s` is null-terminated):

```cpp
char s[10];
std::cin.width(9);
std::cin >> s;
```

Now, only the first 9 characters are read when the user enters string `"Supercalifragilistic"`.

Reading multiple values (buffered input)

The overloaded `>>` functions return a reference to stream object `std::cin` and therefore it is possible to chain together multiple calls to insertion operator `>>` into a single statement:

```cpp
std::cout << "Enter an int, float, double, string: ";
int i;
float f;
double d;
char s[10];
std::cin.width(9);
std::cin >> i >> f >> d >> s;
```

This chaining together of multiple calls is possible because the right-shift operator `>>` is left associative. The chained calls are evaluated as:

```cpp
((((std::cin >> i) >> f) >> d) >> s);
```

with expression `std::cin >> i` evaluating to a reference to `std::cin` with the side effect of extracting characters that form an integer value. This provides expression `std::cin >> f` which will return a reference to `std::cin`, and so on.

Fifth C++ program: `bool` type

Older versions of C didn't have a Boolean type, so we just used integer value 0 to represent *false* and a nonzero value such as 1 to represent *true*. The value 0 evaluates to *false* while any nonzero value evaluates to *true*. C99 introduced the Boolean type using keyword `_Bool`, which can only hold values 0 and 1. Assigning any nonzero value to `_Bool` variable sets it to 1. C99 also introduced `<stdbool.h>` header file which defines macros representing type `bool` and its values `true` and `false`:

```c
#define bool  _Bool
#define true  1
#define false 0
```

Just as with C, any C++ expression that evaluates to zero is considered *false*; if the expression evaluates to a nonzero value it is considered *true*. C++ provides predefined Boolean constants for *true* and *false* values using keywords `true` and `false`, respectively. Such Boolean `true` and `false` values can be represented by variables of built-in data type `bool`. The `bool` type and the `true` and `false` values are convenient for writing predicates (functions that return *true* or *false* results).

```cpp
#include <iostream>

// function prototype required in C++
bool bit_is_set(int value, int position);

int main() {
  int value = 7, bit = 2;
  bool b = bit_is_set(value, bit); // b is a real bool
  if (b == true) {
    std::cout << "Bit " << bit << " of value " << value << " is set\n";
  } else {
    std::cout << "Bit " << bit << " of value " << value << " is NOT set\n";
  }

  int x  = bit_is_set(value, bit); // OK, converting bool to int
  bool c = 42; // possible warning:
               // truncating int to bool (c is true, 1 as int)

  // write to standard output: b is 1, c is 1, x is 1
  std::cout << "b: " << b << " | " << c << " | " << x << '\n';
}

// & is bitwise AND and << is left-shift operator
bool bit_is_set(int value, int position) {
  return (value & (1 << position)) ? true : false;
}
```

Sixth C++ program: Testing input in input stream

What happens if you enter incorrect data when running the program input.out? You'll find out that input from `std::cin` is not well suited for interaction with human users. If a user provides input in the wrong format, your programs can behave in a confusing way. Suppose, for example, that a user types 10.75 as the first input in the program. The 10 will be read and stored in variable `i`. The $.75$ remains in the buffer and will be considered in the next input statement which stores that value in variable `f`. This is not intuitive and probably not what the user expected.

If the user had entered $.75$ as the first value, the user will experience a bigger problem. The buffer contains $.75$ which is not suitable for an integer. Therefore, no input is carried out and zero is stored in variable `i`. What's more, the `std::cin` input stream sets itself to a *failed* state. This means, `std::cin` has lost confidence in the data it receives, so all subsequent extractions result in an immediate return with no value stored.

Recognizing and solving input problems is a necessary skill for building robust programs that can survive untrained or careless users. You can call the error processing functions provided by the input stream to process errors. In particular, you can test for extraction errors on an input stream by calling the `fail` member function of `std::istream`:

```cpp
std::cout << "Enter an integer value: ";
int i;
std::cin >> i;
if (std::cin.fail()) {
  std::cout << "Bad input. End of input.\n";
  // do something to prompt user to provide correct input or exit program
}
```

A better idiom to read a single value from an input stream and test whether the read was successful is to use an `if` statement to test whether the input stream is in a failed state or not:

```cpp
std::cout << "Enter an integer value: ";
int i;
if (std::cin >> i) {
  // if true, integer value was read
} else {
  // bad input was entered and the input stream is in failed state
}
```

Understanding how this code works is a bit subtle. Start by remembering that expression `std::cin >> i` evaluates to a reference to `std::cin`, so that asking for the value of `std::cin >> i` in the `if` statement's condition is equivalent to executing `std::cin >> i` and then asking for the value of `std::cin`. Because `std::cin` has type `std::istream`, which is part of the standard library, we must look to the definition of class `std::istream` for the meaning of `if (cin)`. The details of the member function that implements this behavior is complicated enough that this discussion is delayed until we study class design later in the course. For now, what we need to know is that the `std::istream` class provides a conversion that can be used to convert `std::cin` into a `bool` value that can be used in a conditional expression. Whether the `bool` value is `true` or `false` depends on the internal state of the `std::istream` object, which will remember whether the last attempt to read worked. Thus, using `std::cin` as a condition is equivalent to testing whether the last attempt to read from `std::cin` was successful.

There are several ways in which trying to read from a stream can be unsuccessful:

- We might have encountered input that is incompatible with the type of the variable that we are trying to read, such as might happen if we try to read an `int` value and find something that isn't a number. An example of incorrect input was explained earlier.
- We might have reached the end of the input file that the stream represents.
- The system might have detected a hardware failure on the input device.

In any of these cases, the effect is the same: Using this input stream as a condition will indicate that the condition is `false`. Moreover, once we have failed to read from a stream, all further attempts to read from that stream will fail until we reset the stream, which we'll look at a later point in the course when we consider reading data from files.

The next code example illustrates an useful idiom for recognizing the end of valid values in an input stream or recognizing the end of the input file that the stream represents. The following code in source file `average.cpp` reads an unknown of integer values from standard input and computes the average of these unknown count of values:

```cpp
int main() {
  int sum = 0, count = 0, value;
  // continue reading integers until non-integer value is encountered
  while (std::cin >> value) {
    sum += value; // accumulate sum of integer values
    ++count;      // keep track of how many integer values were encountered
  }
  double average = (double)sum/count; // notice the C-style cast operator
  std::cout << "Average of integer values: " << average << "\n";
}
```

Compile, link, and then run the program, say `average.out` in source file `average.cpp`. The program will wait for the user to enter zero or more integers. You enter an integer value followed by Enter key for the program to process the integer value. Continue this until you've entered all of the integer values that you want the program to process. You can signal to the program to terminate the `while` loop by providing a non-integer value as input. Why? Because (as explained earlier), the `while` statement's condition `std::cin >> value` will evaluate to `std::cin` which in turn will evaluate to `bool` value `false` since `std::cin` will be in a failed state (because the input stream contains a character that is not compatible with an integer causing the read to fail).

```
$ ./average.out
Average of input values: 2
$
```

A second way to terminate the `while` loop is to simulate the End-of-File character using the Ctrl+D keyboard combo. The Ctrl+D combo is a special character in Linux called the End-of-File (EOF) character that signals to programs that no further input is to be expected. You enter a sequence of integers followed by Ctrl+D followed by Enter key. The program will stop reading from standard input as soon as it reads Ctrl+D.

A third way is to collect the integer values to be processed in a text file `input.txt` and redirecting the contents of this file to the input file stream with the input indirection symbol `<`. After reading all the integer values in the file, EOF is signaled to the program.

In all cases, the `while` statement's condition `std::cin >> value` will evaluate to `std::cin` which in turn will evaluate to `bool` value `false` since `std::cin` will be in a failed state (because the input stream contains a character that is not compatible with an integer causing the read to fail).