Lecture 12: Functions and File IOin C++, Preprocessor Step of Compilation

November 5th, 2018

Topics: Functions, Command line arguments and formatting, (file) IO, pre-processor and `#include`.

1 Functions in C++

Functions allow us to decompose a program into smaller components It is easier to implement, test, and debug portions of a program in isolation; further, decomposition allows work to be spread among many people working mostly independently. When done properly, resulting programs are easier to understand and maintain: we seek to eliminate duplicated code and reuse functions across multiple programs. Don't Repeat Yourself (DRY vs. WET code).

```
int sum(int a, int b) {
  int c = a + b;
  return c;
}
```

Components:

```
return_type function_name(argument_type1 argument_var1, ...) {
   // function body
   return return_var; // return_var must have return_type
}
```

Consider `src/sum1.cpp`:.

```
#include <iostream>
int sum(int a, int b) {
  int c = a + b;
  return c;
}

int main() {
  int a = 2, b = 3;
  int c = sum(a,b);
  std::cout << "c = " << c << std::endl;
}
```

We can compile with `g++ -Wall -Wextra -Wconversion sum1.cpp -o sum1` and run:

```
$ ./sum1
c = 5
```

1.1 Order of Declaration Matters

Consider `src/sum2.cpp`: we attempt to *use* a function *before* defining it. This yields an error.

```cpp
#include <iostream>

int main() {
  int a = 2, b = 3;
  // the compiler does not yet know about sum()
  int c = sum(a,b);
  std::cout << "c = " << c << std::endl;
}

int sum(int a, int b) {
  int c = a + b;
  return c;
}
```

The compiler imports the objects defined in `iostream`, but when it gets to the expression `sum(a,b)` it doesn't find an object named `sum` to be defined in the current scope. Output:

```
$ g++ -Wall -Wextra -Wconversion sum2.cpp -o sum2
sum2.cpp: In function 'int main()':
sum2.cpp:7:18: error: 'sum' was not declared in this scope
  int c = sum(a,b);                ^
```

Function Declaration A function *declaration* specifies the function name, input argument type(s), and output type only; it need not specify the implementation (code) for the function, however it does critically specify the information needed from a compiler in order to validate its use within a program.

Function Definition A function *definition* is the code that implements the function, and it is legal to call a function if it has been defined or *simply declared*; see `src/sum3.cpp`.

```cpp
#include <iostream>

// Forward declaration or prototype
int sum(int a, int b);

int main() {
  int a = 2, b = 3;
  int c = sum(a,b);
  std::cout << "c = " << c << std::endl;
}

// Function definition
int sum(int a, int b) {
  int c = a + b;
  return c;
}
```

```
$ g++ -Wall -Wextra -Wconversion sum3.cpp -o sum3
$ ./sum3
c = 5
```

1.2 Functions, Data Types, and Implicit Casting

1.2.1 Arguments (and Return Types) Cast to Match Function Declaration

What happens if we provide arguments "with incorrect type"? It really depends on what sub-routines the compiler finds (i.e. what we've defined and `#include`d).[1] Consider `src/datatypes1.cpp`.

```
#include <iostream>

int sum(int a, int b) {
  return a + b;
}

int main() {
  double a = 2.7, b = 3.8;
  int c = sum(a,b);   // Oops! Our sum() only expects integer args.
  std::cout << "c = " << c << std::endl;
}
```

A sub-routine accepting numeric inputs is found, but implicit casting required Above, the compiler runs into `sum(a,b)` and sees that there is only a single `sum()` function defined. This is good news, however the function happens to require `integer` arguments. We learned previously that although C++ is strongly typed, some implicit casting can occur between numeric types. With warning flags enabled, we are told that variables `a` and `b` are implicitly being cast to `int` such that we can proceed with evaluating `sum(a,b)`. Output:

```
$ g++ -Wall -Wextra -Wconversion datatypes1.cpp -o datatypes1
datatypes1.cpp: In function 'int main()':
datatypes1.cpp:14:18: warning: conversion to 'int' from 'double' may alter its value [-Wconversion
  int c = sum(a,b);
                 ^
datatypes1.cpp:14:18: warning: conversion to 'int' from 'double' may alter its value [-Wconversion
$ ./datatypes1
c = 5
```

Return Value (Implicitly) Cast to Match Function Declaration Consider `src/datatypes2.cpp`:

```
#include <iostream>
int sum(int a, int b) {
  double c = a + b;
  return c; // we are not returning the correct type
}

int main() {
  int a = 2, b = 3;
  int c = sum(a,b);
  std::cout << "c = " << c << std::endl;
}
```

[1] In CME 212, we'll emphasize a bit more details around lookup, which may involve inspecting arguments via argument dependent lookup (and possibly template argument deduction) to disambiguate overloaded functions, and how the compiler determines which sub-routine to execute when a `name` is encountered.

```
$ g++ -Wall -Wextra -Wconversion datatypes2.cpp -o datatypes2
datatypes2.cpp: In function 'int sum(int, int)':
datatypes2.cpp:6:10: warning: conversion to 'int' from 'double' may alter its value [-Wconversion]
  return c;
         ^
$ ./datatypes2
c = 5
```

Explicit Casting Of course, we may explicitly instruct the compiler to obey a cast command, `src/datatypes3.cpp`.

```cpp
#include <iostream>

int sum(int a, int b) {
  double c = a + b;
  return (int)c;
}

int main() {
  double a = 2.7, b = 3.8;
  int c = sum((int)a,(int)b);
  std::cout << "c = " << c << std::endl;
}
```

Output after compiling with `g++ -Wall -Wextra -Wconversion datatypes3.cpp -o datatypes3`:

```
$ ./datatypes3
c = 5
```

1.3 `void` Data Type: Absent/Unspecified

We can think of using the `void` keyword to indicate absence of data, e.g. `src/void1.cpp`.

```cpp
#include <iostream>

void printHeader(void) {
  std::cout << "-------------------------" << std::endl;
  std::cout << "      MySolver v1.0      " << std::endl;
  std::cout << "-------------------------" << std::endl;
}

int main() {
  printHeader();
  return 0;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion void1.cpp -o void1
$ ./void1
-------------------------
      MySolver v1.0
-------------------------
```

1.3.1 `void` Functions *Cannot* Return Data

Attempting to return a non-`void` (or absent) value from a `void` function results in a compiler error; `src/void2.cpp`.

```cpp
#include <iostream>

void printHeader(void) {
  std::cout << "-------------------------" << std::endl;
  std::cout << "      MySolver v1.0      " << std::endl;
  std::cout << "-------------------------" << std::endl;
  return 0;
}

int main() {
  printHeader();
  return 0;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion void2.cpp -o void2
void2.cpp: In function 'void printHeader()':
void2.cpp:8:10: error: return-statement with a value, in function returning 'void' [-fpermi
  return 0;
         ^
```

1.3.2 Explicitly Specifying a `void` Return

We can simply use `return;` to exit from a `void` returning function; e.g. `src/void3.cpp`:

```cpp
#include <iostream>

void printHeader(void) {
  std::cout << "-------------------------" << std::endl;
  std::cout << "      MySolver v1.0      " << std::endl;
  std::cout << "-------------------------" << std::endl;
  return;
}

int main() {
  printHeader();
  return 0;
}
```

Output after compiling with `g++ -Wall -Wextra -Wconversion void3.cpp -o void3`:

```
-------------------------
      MySolver v1.0
-------------------------
```

1.3.3 Ignoring Return Value

This is simply done by not *using* the return value in an assignment or as part of an argument to a function call; `src/ignore.cpp`:

```
#include <iostream>
int sum(int a, int b) {
  int c = a + b;
  return c;
}

int main() {
  int a = 2, b = 3;
  sum(a,b); // legal to ignore return value if you want
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion ignore.cpp -o ignore
$ ./ignore
```

1.4 Function Scope

Functions have their own scope. When a function is called with arguments, a new scope is created wherein the arguments are *copied* into the new environment. Variables outside the scope of the function are not typically accessible; see `src/scope1.cpp`.

```
#include <iostream>

int sum(void) {
  int c = a + b;  // Error: variables used which are not declared in scope.
  return c;
}

int main() {  /* All bets are off; program won't compile... */ }
```

Output:

```
$ g++ -Wall -Wextra -Wconversion scope1.cpp -o scope1
scope1.cpp: In function 'int sum()':
scope1.cpp:5:11: error: 'a' was not declared in this scope
  int c = a + b;
          ^
scope1.cpp:5:15: error: 'b' was not declared in this scope
  int c = a + b;
              ^
...
```

1.5 Global Scope

This is in general discouraged; you should rarely rely on this practice, since if you intend for your program to be re-used by others then global variables can lead to conflicts and correctness bugs; see `src/scope2.cpp`.

```cpp
#include <iostream>

// an be accessed from anywhere in the file (bad, bad, bad!)
int a;

void increment(void) { a++; }

int main() {
  a = 2;
  std::cout << "a = " << a << std::endl;
  increment();
  std::cout << "a = " << a << std::endl;
}
```

It's maybe not surprising that the program outputs `a = 2` followed by `a = 3`. However, the problem here really lies in the fact that variable `a` is being used within `main` and also this very same variable is always being used by our `increment` function; this is likely to lead to a subtle bug at best. Output:

```
$ g++ -Wall -Wextra -Wconversion scope2.cpp -o scope2
$ ./scope2
a = 2
a = 3
```

1.6 Passing Arguments (by *value*)

We strive to be explicit about what the arguments our functions depend on; `src/passing1.cpp`.

```cpp
#include <iostream>

void increment(int a) {
  a++;
  std::cout << "a = " << a << std::endl;
}

int main() {
  int a = 2;

  increment(a);
  std::cout << "a = " << a << std::endl;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion passing1.cpp -o passing1
$ ./passing1
a = 3
a = 2
```

1.6.1 Passing Pointer Arguments (still by *value*)

We must be careful of what is being copied! If we pass a pointer, the value gets copied into a new pointer variable. The result is that "both" pointers reference the same underlying data; `src/passing2.cpp`.

```cpp
#include <iostream>

void increment(int a[2]) {
  a[0]++;
  a[1]++;
}

int main() {
  int a[2] = {2, 3};

  std::cout << "a[0] = " << a[0] << ", " << "a[1] = " << a[1] << std::endl;
  increment(a);
  std::cout << "a[0] = " << a[0] << ", " << "a[1] = " << a[1] << std::endl;
}
```

```
$ g++ -Wall -Wextra -Wconversion passing2.cpp -o passing2
$ ./passing2
a[0] = 2, a[1] = 3
a[0] = 3, a[1] = 4
```

If we *copy* a pointer, the value of the pointer refers to the same underlying data C++ defaults to pass by value, which means that when calling a function the arguments are copied into a new stack-frame. However, you need to be careful and recognize what is being copied! In the case of a number like `int a`, what is being copied is the value of the number, but for a static array like `int a[2]`, what is being passed and copied is the location in memory where the array data is stored.

1.7 Functions and Modularity

We strive to decompose our programs into reusable modules that are easily debugged and inspected. This may mean splitting code across files; perhaps we have many complicated sub-routines we must define in order to run a non-trivial `main` program; in such an instance, it'd be preferable to compartmentalize our code; a simple example `src/main4.cpp`.

```cpp
#include <iostream>

int sum(int a, int b);

int main() {
  int a = 2, b = 3;
  int c = sum(a,b);
}
```

`src/sum4.cpp`:

```
int sum(int a, int b) {
  int c = a + b;
  return c;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion main4.cpp sum4.cpp -o sum4
$ ./sum4
c = 5
```

1.7.1 Linker Errors

We'll discuss in a later lecture the details of the compilation process. One step is to ensure that each function being used has a valid definition. Suppose we have a prototype for a function, but no corresponding definition? The compilation process will error during when trying to link dependencies from the main routine; `src/main5.cpp`.

```
#include <iostream>

int sum(int a, int b);

int main() {
  int a = 2, b = 3;
  int c = sum(a,b);
  std::cout << "c = " << c << std::endl;
}
```

`src/sum5.cpp`:

```
double sum(double a, double b) {
  double c = a + b;
  return c;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion main5.cpp sum5.cpp -o sum5
/tmp/ccCKlsvX.o: In function main':
main5.cpp:(.text+0x21): undefined reference to sum(int, int)'
collect2: error: ld returned 1 exit status
```

2 Command line arguments

Let's consider how we can pass arguments from the command line to a C++ program. We've mentioned previously that `main()` specifies the start of a program, and has a signature like

```
int main(int argc, char *argv[]) { /* body */ }
```

where here

- `argc` is a non-negative value representing the *number* of arguments passed to the program from the calling environment.
- `argv` is a pointer to the first element of an *array of pointers*; each element in the array is a sequence of null-terminated multibyte strings (NTMBS) representing arguments that were passed to the program.[2]

```
#include <iostream>

int main(int argc, char *argv[]) {
  // Display the command line arguments
  for (int n = 0; n < argc; n++) {
    std::cout << n << " " << argv[n] << std::endl;
  }
  return 0;
}
```

Output:

```
$ ./argv1 hello.txt 3.14 42
0 ./argv1
1 hello.txt
2 3.14
3 42
```

2.1 Formatting (a Minimum Number of) Command Line Arguments

If our program requires a minimum number of inputs in order to execute, we must reflect this in the control flow of the program. Notice that in the following example, we simply print a helpful usage message to console and exit the program (without returning an error).

```
#include <iostream>
#include <string>

int main(int argc, char *argv[]) {
  // Catch the case where insufficient arguments are provided.
  // The first argument is the name of the executable being run!
  if (argc < 4) {
    std::cout << "Usage:" << std::endl;
    std::cout << " " << argv[0] << " <filename> <param1> <param2>" << std::endl;
    return 0;
  }
  // We're now guaranteed that three parameters were passed as argument to the program.
  std::string filename = argv[1];       // Strings can be initialized with NTMBS.
  double param1 = std::stof(argv[2]);   // String-to-Float.
  int param2    = std::stoi(argv[3]);   // String-to-Integer.

  std::cout << "filename = " << filename << std::endl;
  std::cout << "param1 = "   << param1   << std::endl;
  std::cout << "param2 = "   << param2   << std::endl;
}
```

[2] We are guaranteed that `argv[argc]` is a null pointer.

Note that all the command line arguments are treated as `char*`s, whence we must cast the underlying data to the appropriate type. We've taken advantage of the fact that the compiler doesn't care about white-space to line up syntactically similar statements to reflect their commonalities. Output:

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra argv2.cpp -o argv2
$ ./argv2 hello.txt 3.14 42
filename = hello.txt
param1 = 3.14
param2 = 42
```

3 IO in C++

3.1 Default IOStream Settings

Without considering formatting options, the default iostream settings consider precision when formatting floating point values. The default settings are to print up to six digits (including both integer and fractional digits), but to omit trailing zeros if the result can be precisely displayed.

```cpp
#include <iostream>
int main() {
  double a = 2.;
  std::cout << "a = " << a << std::endl;
}
```

```
$ ./formatting1
a = 2
```

Even though `a` is type `double`, it has no fractional part, so the default formatting options encourage only the minimum number of digits to be displayed accurately.

3.2 Manipulators and Formatting Flags

Just like Python, a significant amount of flexibility in string formatting is offered.

3.2.1 Manipulators

Manipulators are helper functions that allow us to control how input and output streams behave. E.g. displaying numeric values in different base-representations (decimal, octal, hexadecimal) alignment (left or right), and case-sensitivity (upper-case).

```cpp
#include <iostream>
int main() {
    std::cout << "The decimal number 42 in decimal: " << std::dec << 42 << std::endl
              << "The decimal number 42 in octal:   " << std::oct << 42 << std::endl
              << "The decimal number 42 in hex:     " << std::hex << 42 << std::endl;
}
```

Compiling and running, we see that

```
 g++ -std=c++11 formatting_flags.cpp -o formatting_flags
./formatting_flags
The decimal number 42 in decimal: 42
The decimal number 42 in octal:   52
The decimal number 42 in hex:     2a
```

This should seem familiar, perhaps after we recall that in hex A-F maps to {10, ..., 15}.

$$42 = \mathbf{4} \times 10^1 + \mathbf{2} \times 10^0 = \mathbf{5} \times 8^1 + \mathbf{2} \times 8^0 = \mathbf{2} \times 16^1 + \mathbf{10} \times 16^0.$$

3.2.2 Formatting Flags and `setf`

We can apply manipulators as we did above, *or* we can use the `setf` (and implicit Boolean operations) together with a *formatting flag* (analogous to a manipulator) to obtain a similar result. Akin to each manipulator, there are formatting flags in the standard library which represent the *state* of a stream's formatting options; they can be implemented as a bitmask datatype, and so using `setf(flags)` can be thought of as effectively using bitwise operations to set and clear relevant states.[3] So, we could equivalently re-write our program above, replacing manipulators (e.g. `std::dec`) with formatting flags (e.g. `std::ios_base::dec`).

```
#include <iostream>
#include <iomanip>
 int main() {
    std::cout.setf(std::ios_base::dec);
    std::cout << "The number 42 in decimal: " << 42 << std::endl;
    // Here, we set octal option, but first clear any formatting already specified.
    std::cout.setf(std::ios_base::oct, std::ios_base::basefield);
    std::cout << "The number 42 in octal:   " << 42 << std::endl;
    std::cout.setf(std::ios_base::hex, std::ios_base::basefield);
    std::cout << "The number 42 in hex:     " << 42 << std::endl;
}
```

`setf` Updates Internal Flags (with Boolean Logic) Let `fl` denote the state of our internal formatting flags. Then, the one-argument `setf` sub-routine simply applies the bit-mask with something like `fl = fl | flags`, whereas the two-argument `setf` sub-routine first clears bits specified by the mask (second argument) and *then* sets the (cleared) flags to those specified by first argument with something like
`fl = (fl & ~mask) | (flags & mask)`.

3.3 Example Usage of Common Manipulators

3.3.1 Unconditionally Show Decimal Point via `showpoint`

The `showpoint` manipulator specifies to unconditionally display fractional parts of floats.

[3] Bitwise operators include unary negation (~), binary infix `and` (&), `or` (|), and `XOR` (^).

```
#include <iostream>

int main() {
  double a = 2., b = 3.14, c = 1.23456789, d = 12.3456789;
  int e = 4;

  std::cout.setf(std::ios::showpoint);
  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;
  std::cout << "c = " << c << std::endl;
  std::cout << "d = " << d << std::endl;
  std::cout << "e = " << e << std::endl;

  return 0;
}
```

Notice that in the following output, floating point representations always include six digits (integer and fractional part included); since `int`s don't have a fractional part, the `showpoint` option doesn't apply.

```
$ ./formatting3
a = 2.00000
b = 3.14000
c = 1.23457
d = 12.3457
e = 4
```

Notice that in the event that more than six digits of precision are required, that the displayed value is rounded accordingly.

3.3.2 Controlling Decimal Places

We can get fixed-width formatting using `fixed`. The following example always displays three decimal places for any floating point datatype, inclusive of trailing zeros; see `formatting4.cpp`.

```
#include <iostream>
int main() {
  double a = 2., b = 3.14;
  int c = 4;
  //Always show 3 decimal places
  std::cout.setf(std::ios_base::fixed, std::ios_base::floatfield);
  std::cout.setf(std::ios_base::showpoint);
  std::cout.precision(3);
  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;
  std::cout << "c = " << c << std::endl;  # Integers not affected by options re: fractional parts.
}
```

The above example uses formatting flags, but of course we could have used manipulators.

3.3.3 Scientific Notation

```
int main() {
  double a = 2., b = 3.14;
  int c = 4;
  std::cout.setf(std::ios::scientific, std::ios::floatfield);
  std::cout.precision(3);
  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;
  std::cout << "c = " << c << std::endl;
}
```

Output shows `a = 2.000e+00`, `b = 3.140e+00`, and `c = 4`.

3.3.4 Field Width

In the context of output, the `std::ios_base::width()` function manages the minimum number of characters to generate on output operations.

```
#include <iostream>
int main() {
  std::cout << "Minimum Field width...currently set to " <<  std::cout.width() << '\n';
  std::cout << "        10        20        30\n";
  std::cout << "         ^         ^         ^\n";
  std::cout << "         |         |         |\n";
  std::cout << "123456789-123456789-123456789|\n";
  std::cout << 12.345 << std::endl;
  std::cout.width(15);
  std::cout << 12.345 << std::endl;
  std::cout.width(30);
  std::cout << 12.345 << std::endl;
}
```

```
$ ./formatting6
Minimum Field width...currently set to 0
        10        20        30
         ^         ^         ^
         |         |         |
123456789-123456789-123456789|
12.345
         12.345
                        12.345
```

3.3.5 Fill character

What if instead of a minimum output field width (via `std::ios_base::width()` as above), we wanted to specify an exact field width? For this, we can use the manipulator `std::setw(int)`.

```
#include <iomanip>
#include <iostream>
 int main() {
  std::cout.fill('0');
  for(int n = 0; n < 10; n++)
    std::cout << std::setw(n < 5 ? 2 : n) << n << std::endl;
}
```

Recall the ternary operator (i.e. the conditional operator). Output:

```
$ ./formatting7
00
...
04
00005
000006
0000007
00000008
000000009
```

3.4 File IO

```cpp
#include <iostream>
#include <fstream>
int main() {
  double a = 2., b = 3.14;
  int c = 4;
  std::ofstream f("formatting.txt");
  f.setf(std::ios::showpoint);
  f << "a = " << a << std::endl;
  f << "b = " << b << std::endl;
  f << "c = " << c << std::endl;
  f.close();
}
```

Output:

```
$ ./formatting8
$ cat formatting.txt
a = 2.00000
b = 3.14000
c = 4
```

3.5 Examples of File IO in C++

3.5.1 Loading a Tabular Dataset - Homogeneous Data

Remember the Movielens data? Each row contains exactly four integers separated each by at least one space.

```
$ cat u.data
196 242 3   881250949
186 302 3   891717742
...
6   86  3   883603013
```

```cpp
#include <fstream>
#include <iostream>

int main() {
  std::ifstream f;
  f.open("u.data");
  if (f.is_open()) {
    int uid, mid, rating, time;
    while (f >> uid >> mid >> rating >> time) {
      std::cout << "user = "     << uid;
      std::cout << ", movie = "  << mid;
      std::cout << ", rating = " << rating << std::endl;
    }
    f.close();
  }
  else {
    std::cerr << "ERROR: Failed to open file" << std::endl;
  }
  return 0;
}
```

After compiling.

```
$ ./file1
user = 196, movie = 242, rating = 3
user = 186, movie = 302, rating = 3
...
user = 305, movie = 451, rating = 3
user = 6, movie = 86, rating = 3
```

There are actually a *lot* of non-trivial implementation details here.

3.5.2 Revisiting `operator>>`

See `src/file1.cpp`. Recall how the stream insertion `operator>>` behaves: it returns a reference to the remaining stream after reading a unit of data from the stream. Basic signature:

```cpp
basic_istream& operator>>( int& value );
```

Return Type is a Reference to an Input Stream This signature tells us that the output type is a reference to (denoted by the trailing`&` after the type) a `basic_istream` object, and that the input argument accepts an integer by reference (i.e. a variable we wish to mutate). I.e. if we have an expression like `f >> uid`, then we can think of this as calling the `operator>>` *method* on our input-stream *object*, with `uid` provided as argument. Whence `f >> uid >> mid >> rating >> time` first takes the file-stream referred by `f`, reads data into `uid` and returns the remaining stream. We then repeat, this time reading the next datum and placing its contents into `mid`, etc.

There are two details I've glossed over: how do we know the number of white-spaces to skip, and how do we determine how many characters to read from the input stream?

Formatted Input Functions and `std::ios_base::skipws` You may be wondering what kind of logic is used in parsing the white-spaces from our data, e.g. what happens if there is a varying number of white-space characters between fields? The insertion stream operator `std::operator>>` is a Formatted Input Function adhering to several specifications – one of which determines how to handle white-spaces i.e. the formatting flag `ios_base::skipws` for the input stream is inspected and if set to `true` then (any arbitrary number of) white-spaces are skipped! The logic is to extract and discard characters that are white-space until either a non-white-space character is read *or* the end of the file is reached.

Determining # Characters to Read via `num_get` when we call `operator>>(int&)` (which is made explicit by the type of the argument appearing on the right hand side of `>>`), the method is instructed to read data from input stream and store the result in an `integer` object. Implicitly, the sub-routine calls `num_get` which reads the *appropriate* number of characters from the input stream: the idea is that we read characters such as `0-9`, `A-F`, and select characters like `.` from the stream and accumulate the results until we run into a character that is *not* associated with a numeric representation of data.

Determining # Lines to Read via IO State Flags Lastly, we comment on the use of the expression `while(std::cin >> ...)`: what guides the control-flow, i.e. how can the expression be evaluated in a way which makes sense to use in a predicate statement? It turns out that `std::cin` keeps track of `std::ios_base::iostate` and specifically whether the last extraction succeeded or whether an end-of-file bit has been encountered.

3.5.3 Heterogeneous Data Types on each Line

See `src/dist.female.first`:

```
MARY           2.629  2.629      1
PATRICIA       1.073  3.702      2
...
KAREN          0.667 12.742     13
BETTY          0.666 13.408     14
```

Such a file requires a little bit more care in order to parse properly. Note the data types!

```cpp
std::ifstream f;

f.open("dist.female.first");
if (f.is_open()) {
  std::string name;
  double perc1, perc2;
  int rank;
  while (f >> name >> perc1 >> perc2 >> rank) {
    std::cout << name << ", " << perc1 << std::endl;
  }
  f.close();
}
else { std::cerr << "ERROR: Failed to open file" << std::endl; }
```

3.5.4 Varying # of Columns/Variables in each Line/Observation

What if lines have a varying amount of data to load? E.g. a circle may be defined by an $(x, y, \texttt{radius})$ triplet, whereas a line segment, triangle, and rectangle require that we specify the two, three, or four (x, y) corner points respectively.

```
$ cat geometry1.txt
workspace 0 0 10 10
circle 3 7 1
triangle 4 6 8 6 5 7
rectangle 1 1 8 2
```

```cpp
f.open(filename);
if (f.is_open()) {
  std::string shape;
  while (f >> shape) {
    int nval;
    // Determine the shape and how many values need to be read
    if (shape == "workspace" or shape == "rectangle") nval = 4;
    else if (shape == "circle")                       nval = 3;
    else if (shape == "triangle")                     nval = 6;
    else {
      std::cerr << "ERROR: Unknown shape '" << shape << "'" << std::endl;
      return 1;
    }
  float val[6];
  for (int n = 0; n < nval; n++)     // Read appropriate number of values
    f >> val[n];
```

Notice that in the last `for`-loop, we are using `nval` as our loop-criterion such that we are careful to read the appropriate number of data points depending on the geometry; see `src/file4.cpp`.

3.5.5 Read an Entire Line at a Time via `getline()`

There is a `getline()` function to read a line at a time; it defaults to treating a newline character `\n` as the delimiter, but this can be overridden with a secondary argument.

```cpp
f.open(filename);
if (f.is_open()) {
  std::string line;
  while (getline(f, line))
    std::cout << line << std::endl;
  f.close();
}
else {
  std::cerr << "ERROR: Failed to open file" << std::endl;
}
```

4 String Stream

A `std::basic_stringstream` offers a convenient way to manipulate strings in a way similar to what we learned above with IO devices.

4.1 Using `stringstream` to Enable IO-Like Operations on `strings`

Example: Simplify a Directory Path Suppose you're given a Unix-style absolute path, and are asked to simplify it. E.g.

- `/myfolder/` $\rightsquigarrow$ `/myfolder`, or
- `/a/./b/../../c/` $\rightsquigarrow$ `/c`, and perhaps lastly
- `/a//b////c/d//././/..` $\rightsquigarrow$ `/a/b/c`.

There are many solutions. One might attempt to split the input string based on the path delimiter (`/`) and then for each folder (string) returned, consider whether we are applying a reversal (via `..`), a nullpotent operator (one of `.` or the empty string `""`), or if we're actually specifying a new sub-folder (i.e. any other sequence of characters). An implementation may use a `stringstream` (and perhaps `vector`, which we'll learn about soon). Consider `simplifyPath.cpp`, and realize that when we use `std::getline()`, that it also can be used in a predicate since it utilizes `std::ios_base::iostate` state flags.

```
std::string simplifyPath(std::string path) {
  std::string res, tmp;
  std::vector<std::string> stk;
  std::stringstream ss(path);
  while(getline(ss,tmp,'/')) {
    if (tmp == "" or tmp == ".")       continue;
    if (tmp == ".." and !stk.empty()) stk.pop_back();
    else if (tmp != "..")             stk.push_back(tmp);
  }
  for(auto str : stk) res += "/" + str;
  return res.empty() ? "/" : res;
}
```

Advantage of `stringstream` In this situation, one might ask: "why not simply take the string `path` and manually split on `/`?". Well, in order to do this in C++ we'd need to do something like: (i) use `str.find()` twice to search for a *pair* of delimiting `/`, after which we (ii) extract the sequence of characters in between, and continue until no more delimiting `/` are found in the string. Implementing (i) and (ii) requires a few lines of code, and it's much easier to simply treat the string `path` as a `stringstream`, wherein we can use `getline` with a secondary argument set to `/`; the `stringstream` abstraction yields a more concise program.

5 The preprocessor and #include

We have used functionality from the C++ standard library for output to the screen using `cout`, performing I/O with files, using the string object, etc. We've mentioned that a library is a collection of functions, data types, constants, class definitions, etc., somewhat analogous to a Python module. At a minimum, accessing the functionality of a library requires `#include` statements. What's going on here?

5.1 #include

So what actually happens when you put something like `#include <iostream>` in your file? `<iostream>` is a way of referring to a file called iostream that is part of the compiler installation and on the corn machines is found at `/usr/include/c++/4.8/iostream`. These types of files are called include or header files and contains forward declarations (prototypes) of functions, class definitions, constants, etc.

Preprocessor: "Glorified Copy-Paste" Before files are processed by the compiler, they are run through the C preprocessor, `cpp`. What does the preprocessor do? For one thing it processes those `#include` statements.

Hacking Around with the Preprocessor

```
$ cat hello.txt
Hello!
$ cat goodbye.txt
#include "hello.txt"
Goodbye!

$ cpp -P goodbye.txt
Hello!

Goodbye!
```

5.2 Compilation Process

5.2.1 Standard Decomposition of a Program

- Function (and type) *declarations* go in header (`.hpp`) files.
- Function *definitions* go in source (`.cpp`) files.
- Source files that want to use the functions must `#include` the header.

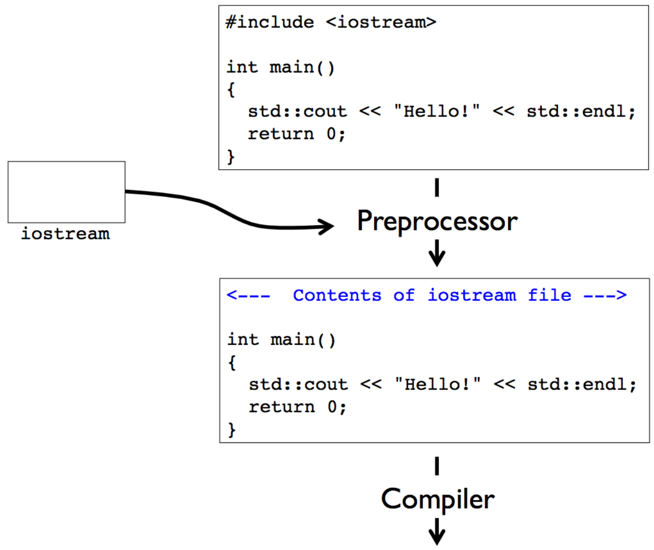

Figure 1: When we `#include` a library, the contents are effectively inserted into our source code before compilation continues.

`src/main6.cpp`:

```
#include <iostream> #include "sum6.hpp"
 int main() {
  double a = 2, b = 3;    double c = sum(a,b);
  std::cout << "c = " << c << std::endl;
}
```

Then, within `src/sum6.hpp` we simply include the *prototype* or declaration of the function.

```
double sum(double a, double b);
```

Lastly, within `src/sum6.cpp` we actually implement the function *definition*. Realize that we `#include` our `.hpp` file here to ensure it gets used in the compilation process.

```
#include "sum6.hpp"

double sum(double a, double b) {
  double c = a + b;
  return c;
}
```

Now, when we go to compile our `main` program, we must specify that we have a dependency on code contained in `sum6.cpp`. Output:

```
$ g++ -Wall -Wextra -Wconversion main6.cpp sum6.cpp -o sum6
$ ./sum6
c = 5
```

5.2.2 `#include` Syntax

- The `.hpp` file extension denotes a C++ header file
- < > around the file name means that the preprocessor should search for an include file in a system dependent or default directory
- These are typically include files that come with the compiler like `iostream`, `fstream`, `string`, etc.
- Usually these files are somewhere in `/usr/include` with the GNU compilers on Linux
- `"header.hpp"` means that the preprocessor should first search in the user directory, followed by a search in a system dependent or default directory if necessary

6 `#define` and Macros

6.1 `#define`: A Mini Copy-Paste

We can use `#define` such that anytime the symbol is encountered within our source code, it gets replaced by a different sequence of characters; e.g. `src/define1.cpp`.

```
// define ni and nj to be 16
#define ni 16
#define nj 16

int main() {
  int a[ni][nj];
  for(int i = 0; i < ni; i++) {
    for(int j = 0; j < nj; j++) {
      a[i][j] = 1;
    }
  }
}
```

Pass the code through the preprocessor, and observe that instances of `ni` and `nj` have been *replaced* by the value 16, and that comments have been removed!

```
$ cpp -P define1.cpp
int main() {
  int a[16][16];
  for(int i = 0; i < 16; i++) {
    for(int j = 0; j < 16; j++) {
      a[i][j] = 1;
    }
  }
}
```

6.2 Macros

The real power of `#define` is in setting up macros. Similar to functions but handled by the preprocessor! They don't involve overhead in copying arguments into a temporary stackframe which must be set-up and torn-down; `src/define2.cpp`.[4]

```
#include <iostream>

#define sqr(n) (n)*(n)

int main() {
  int a = 2;

  int b = sqr(a);
  std::cout << "b = " << b << std::endl;

  return 0;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion define2.cpp -o define2
$ ./define2
b = 4
```

Be Careful: We Really Meant Ctrl-V The `#define` really is a copy-paste! I.e. we can omit parentheses at the risk of screwing up our order of operations; `src/define3.cpp`.

```
#include <iostream>

#define sqr(n) n*n

int main() {
  int a = 2;

  int b = sqr(a+3);
  std::cout << "b = " << b << std::endl;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion define3.cpp -o define3
$ ./define3
b = 11
```

At this point, you can guess why you can't have an in-line comment following a macro definition!

[4]Note that the functionality of Macros for this purpose is largely superceded by compiler optimizations of inlining functions.

6.2.1 Predefined Macros

Several are quite useful, and reminiscent of what we've encountered in Python; `src/define4.cpp`.

```
#include <iostream>

int main() {
  std::cout << "This line is in file " << __FILE__
            << ", line " << __LINE__ << std::endl;
  return 0;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion define4.cpp -o define4
$ ./define4
This line is in file define4.cpp, line 5
```

6.3 Conditional compilation

We can use the preprocessor to support conditional compilation using `ifdef` macros; `src/conditional.cpp`.

```
#include <iostream>

#define na 4

int main() {
  int a[na];

  a[0] = 2;
  for (int n = 1; n < na; n++) a[n] = a[n-1] + 1;

#ifdef DEBUG
  // Only kept by preprocessor if DEBUG defined
  for (int n = 0; n < na; n++) {
    std::cout << "a[" << n << "] = " << a[n] << std::endl;
  }
#endif

  return 0;
}
```

Output:

```
$ g++ -Wall -Wextra -Wconversion conditional.cpp -o conditional
$ ./conditional
$ g++ -Wall -Wextra -Wconversion conditional.cpp -o conditional -DDEBUG
$ ./conditional
a[0] = 2
a[1] = 3
a[2] = 4
a[3] = 5
```

7 Reading

- **C++ Primer, Fifth Edition** by Lippman et al.
- Chapter 6: Functions: Sections 6.1 - 6.3
- Chapter 8: The IO Library
- Chapter 17: Specialized Library Facilities: Section 17.5.1
- String formatting: `http://umich.edu/~eecs381/handouts/formatting.pdf`