Selected Slides from the Koffman-Wolfgang C++ Primer (with spot-edits and additions by KWR and AH)

From the Instructor Materials slides by Elliot Koffman and Paul Wolfgang, copyright © John Wiley and Sons, 2006—2009.

This material is used by permission of John Wiley & Sons, Inc

For private use only by students in CSE250, Fall 2009. Re-distribution or public posting in any form is expressly forbidden.

A Simple Program

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

using namespace std;

int main()
{
  cout << "Enter your name\n";
  string name;
  getline(cin, name);
  cout << "Hello " << name
       << " - welcome to C++\n";
  return 0;
}
```

The `#include` Directive

- The first two lines:

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

  incorporate the declarations of the `iostream` and `string` libraries into the source code.
- If your program is going to use a member of the standard library, the appropriate header file must be included at the beginning of the source code file.

The `using` Statement

- The line

```
using namespace std;
```

  tells the compiler to make all names in the predefined namespace std available.
- The C++ standard library is defined within this namespace.
- Incorporating the statement

```
using namespace std;
```

  is an easy way to get access to the standard library.
  - But, it can lead to complications in larger programs.

The using declaration

- Instead of incorporating all names from a namespace into your program
  - It is a better approach to incorporate only the names you are going to use.
  - This is done with individual using declarations.

```
using std::cin;
using std::cout;
using std::string;
using std::getline;
```

The function main

- Each program must include a main function.
- This function is defined as follows:

```
int main()
{
   …
}
```

where the code for the function appears between the `{` and the `}`.

YMMV Note (by us)

- Some (old) compilers or authorities may want you to say `int main(void) { ...`
  - (when you are not using command-line parameters, that is)
- Some desire or require an explicit `return 0;` at the end of `main`.
- Etc. etc. *If one of the programs we provide does not compile as-is on your home system, please let us know!*

The stream insertion operator

- The statement:

```
cout << "Enter your name\n";
```

  inserts the string into the standard output stream.
  - The result is that it is displayed on the console.

The getline function (ed. by us)

- The statement

```
getline(cin, name);
```

  reads the characters from the input stream (keyboard) until a new line character is entered.
  - The resulting string is stored in the `string name`.
  - getline consumes the \n but does not return it.

The insertion operator again

- The statement:

```
cout << "Hello " << name << " - welcome to C++\n";
```

  outputs three strings to the console:

```
Hello
```

the entered line

```
- welcome to C++
```

- If the characters `John Doe` were entered, the result would be

```
Hello John Doe - welcome to C++
```

Comments (ed. by us)

- There are three types of comments:
- Form 1:
  - A comment begins with the characters `/*` and ends with the characters `*/`
- Form 2:
  - A comment begins with the characters `//` and ends at the end of the current line
- Form 3, using the preprocessor:
  - Begin with `#if(0)`, put a line of dashes for clarity, and after the block to comment-out, put a line of dashes, and the line `#endif`
- All characters of a comment are replaced by a single space by the preprocessor.
- Note that a `/*` that follows a `//` is ignored.
- A `//` that appears after a `/*` is also ignored.
- A comment that begins with a `/*` is terminated by the **first** `*/` that is encountered. This is why one needs Form 3 to comment out blocks of code that may have /*…*/ in it.

More on #include directive

- The #include directive has two forms:

  `#include <`*header*`>`

  – is reserved for standard library headers.

  `#include "`*file-name*`"`

  – is used for user-defined include files.
- The convention is that user-defined include files will end with the extension `.h`.
- Note that the standard library headers do not end with `.h`.

Compiling and Executing

- The command to compile is dependent upon the compiler and operating system.
- For the gcc compiler (popular on Linux) the command would be:
  - g++ -o HelloWorld HelloWorld.cpp
- To execute the program you would then issue the command

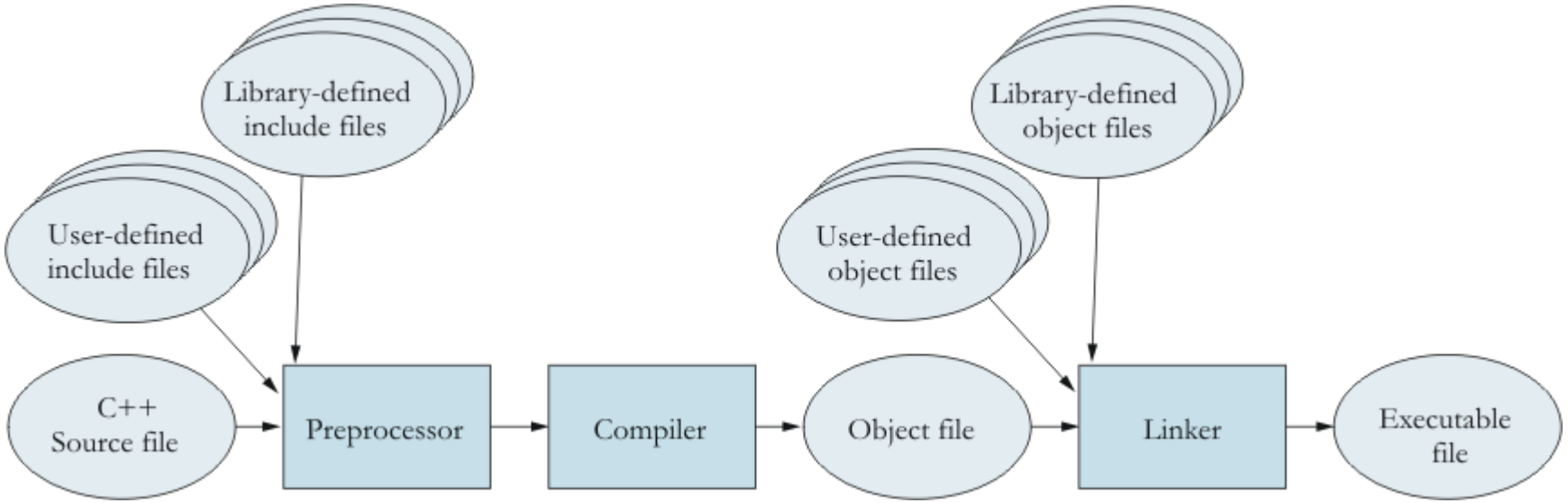
  HelloWorld

More complicated projects require linking. An IDE such as Eclipse can manage this automatically, but it is important to learn how to do it from a console command line, and this uses the “make” / makefile utility.

Compiling and Linking

- A C++ program consists of one or more source files.
- Source files contain function and class declarations and definitions.
  - Files that contain only declarations are incorporated into the source files that need them when they are compiled.
    - Thus they are called *include files*.
  - Files that contain definitions are translated by the compiler into an intermediate form called *object files*.
  - One or more object files are combined with to form the executable file by the linker.

Compiling and Linking



Compiling and Executing

- The command to compile is dependent upon the compiler and operating system.
- For the gcc compiler (popular on Linux) the command would be:

  g++ -o HelloWorld HelloWorld.cpp

- To execute the program you would then issue the command

  HelloWorld

- Confusing? Use the -c option to produce a .o file, the -o option for an executable.

Example of Manual Linking

`g++ -c StringWrap.cpp`

This produces StringWrap.o

`g++ -c SeqClient.cpp`

This produces SeqClient.o

`g++ -o “seqclient” StringWrap.oSeqClient.o`

Produces the final executable seqclient

Use `./seqclientConstitution.txt 1000` to run.

File Streams---simple way (by us)

- File I/O uses the same stream notation as screen I/O with `cout << …` and `cin >> …`
- OK to use simple (value) notation provided you know the file(name) to open right away.
- Need #include<fstream> to get the ifstream and ofstream classes.
- Can open a file `foo.txt` for input by

```
ifstream INPUT("foo.txt", ios::in);
```

File Streams---cont'd.

- To open a file in append mode, do `ofstream OUTPUT("bar.txt", ios::app);`

- Style: ALL-CAPS for unusual user-defined items?
- `std::ios::app` is the "append object" in the `ios` sub-namespace.
- Can read a whitespace-separated item by `INPUT >> item`, and write a value x to a file by `OUTPUT << x`, just as with `cout`.
- Can use `getline(INPUT,myline)` to read the next line into the string variable `myline`.

Reading all input from a file (by us)

- `while (INPUT >> item){…` will terminate after the last item in the file has been read.
- This is a common "walk and chew gum" idiom. It works because the `>>` operator returns a null reference when the stream is exhausted. Better than testing for `eof()`.
- `while(getline(INPUT,myline)) {…` works line-by-line. No `\n` returned in `myline`.
- Still need `INPUT.close()` etc. to close files.

Header and Body Files (by us)

- The purpose of having separate .h and .cpp files is ***to maximize the opportunity for separate compilation***. (And for readability.)
- Java does separate compilation to the max! (.class file is like C/C++ .o object file)
- It is OK to have bodies in a .h file (especially *inside* class braces), so long as they are not too big and don't cause many dependencies.
- Outside class braces, bodies in .h files should be marked **`inline`** (more on this later).

.h and .cpp (by us)

- Header files universally use the extension `.h`, e.g. "`Foo.h`".
- Other files can include the header via `#include "Foo.h"`, with quotes not <…> because this is relative to the current directory.
- AOK to have code in subfolders, e.g. `#include "ISR/AbsISR.h"`
- Body and end-client files (with main) will use `.cpp` in this course (`.cc`, `.cxx` also seen)

Templates Will Be .h Only (by us)

- Template classes will have .h files only, and will not be compiled separately.
  - Our text is like that, ditto earlier CSE250's.
  - Reason not to use separate .cpp for templates is a linking issue, not supported by g++. (Your home IDE/compiler might allow…)
- See example files beginning with "Link…" in the ~regan/cse250/Java2C++ directory.

`extern` and `static` (by us)

- A non-const object declared outside class braces in a .h file should have the keyword `extern` in front.
- Then define/construct it in the .cpp file, without saying “extern”.
- Functions/operators are automatically extern, so they don’t need the keyword.
- (Text mentions `extern "C"` in a conditional-compilation context, which is different, ignorable…)

`extern` and `static` (cont'd)

- For class fields and methods, `static` has the same meaning as in Java.
- But non-const static fields *cannot* be initialized inside the class, nor by inline constructor---not anywhere in the .h file!
- So you could have a Foo.cpp file for a class Foo that has nothing except a few definitions of non-const static fields.
- Field can't be both `static` and `extern`.

The Preprocessor

- The compiler (effectively) makes several passes through the source program.
- The first of these passes is done by what's known as the preprocessor.
    1. Replace trigraphs with their equivalent
    2. Splice long lines into a single line.
    3. Remove comments and replace by a single space.
    4. Split the input file into tokens
    5. Carry out preprocessing directives
    6. Expand macros
- Note that the preprocessor is inherited from the C programming language.

Splicing Long Lines (ed. by us)

- If a line ends with the character \
  - Then the following line is appended to this line and the result is considered a single line.
- Used for long C-style strings, but with <string>, prefer using + operators.
- General: keep lines within 80 columns!
- Break long expressions across lines so that an operator begins the next line.

Macros (ed. by us)

- Macros are defined by the forms:

  `#define` *macro-name macro-definition*

  `#define` *macro-name*`(`*parameters*`)` *macro-definition*

  – Definition ends at the end of the current line.

  – Macros requiring longer lines use long-line splicing.

- Examples:

```
#define NULL 0                                  //AOK, done by default
#define MAX(x, y) ((x) > (y) ? (x) : (y))   //common, but…
```

- Within the program, wherever a macro appears, it is ***textually*** replaced by its definition.
- Without all the parens, MAX could cause errors…
- Hence best to use macros with `#ifndef` etc. only.

Conditional Compilation

- Forms:

```
#ifdef macro-name
code to be compiled if macro-name is defined
#else
code to be compiled if macro-name is not defined
#endif
```

or

```
#ifndef macro-name
code to be compiled if macro-name is not defined
#else
code to be compiled if macro-name is defined
#endif
```

Using Conditional Compilation

- Some functions are defined to be used by both C and C++ programs.
- If a C/C++ compiler is compiling a program as a C++ program, then the macro __cplusplus is defined. (Note the two _ chars).
- Then the function would be declared as follows:

```
#ifdef __cplusplus    //__GNUG__ is for our g++
extern "C" {
#endif
```

function declaration

```
#ifdef __cplusplus
}
#endif
```

Preventing Multiple Includes

- A header file may be included by another header file.
- The user of the header file may not know this and may include a duplicate.
- This may lead to a compile error.
- To prevent this, each include file should be structured as follows:

```
#ifndef unique-name
#define unique-name
…
#endif
```

- Generally *unique-name* is related to the file name.
  - Example `myfile.h` would use the name `MYFILE_H_`

More on #include directive

- The #include directive has two forms:

  `#include <`*header*`>`

  – is reserved for standard library headers.

  `#include "`*file-name*`"`

  – is used for user-defined include files.
- The convention is that user-defined include files will end with the extension `.h`.
- Note that the standard library headers do not end with `.h`.

Rules for Inclusion (by us)

- A *declaration* may not appear twice in the same separate compilation path.
- A *definition* (i.e., value or body) may not appear twice in the whole program…
- …***unless*** it is within class braces---all compilers allow including .h files with bodies in separate compilation units that get linked together.

Using braces and indentation

- There are several coding styles.
- The one used in this text is:
  - Place a `{` on the same line as the condition for an `if`, `while`, or `for` statement.
  - Indent each line of the controlled compound statement.
  - Place the closing `}` on its own line, indented at the same level as the if, while, or for.
  - For **else** conditions, use the form:

```
} else {
```

- But, AOK to align braces vertical a-la 115-116.

Nested If Statements

- If there are multiple alternatives being selected,
  - the **`if`** that appears within an **`else`** part should be on the same line as the **`else`**.
  - Example:

```
if (operator == '+') {
  result = x + y;
  add_op++;
} else if (operator == '-') {
  result = x - y;
  subtract_op++;
}
```

Command-Line Arguments (by us)

- `int main(int argc, char** argv)`, or can do
- `int main(int argc, char* argv[])`
- Invoke as `a.out arg1 arg2 ...`
- In C++, unlike Java, `argv[0]` is the name of the program, as a `char*` type "C-string".
- So `argv[1]`, `argv[2]`,... are the actual args
- (Please otherwise avoid C arrays and strings.)

Giving Filenames as Arguments (by us)

File `stringsorts.cpp` has a basic protocol:

```
…
if (argc >= 2) {   //note *not* an "else" of first "if"
   infileName = args->at(1);
   infile.open(infileName.c_str(), ios_base::in);
   if (! infile.is_open()) {
      cerr << "Unable to open file: " << infileName << endl;
      return(1);
   }
} else {    //argc == 1, i.e. no arguments given
   cerr << "Usage: stringsorts file n   with n >= 0" << endl;
   return(1);
}
```

(Earlier part of file uses a string stream to input an integer more easily.)

Makefiles (4 slides by us)

- A Makefile can be used to manage projects in a number of ways.
- Makefiles can also vary in complexity. They can simply contain lines to compile the code files, use macros, and perform tasks such as cleanup of files generated during compilation (usually used for bigger projects).
- The Java2C++ directory and others under ~regan/cse250/… have examples.

Makefiles

- A Makefile to compile the HelloWorld program could look something like this:

```
HelloWorld: HelloWorld.cpp
        g++ -o HelloWorld HelloWorld.cpp
```

- To compile the code, you need to go to the directory of the Makefile and HelloWorld.cpp and execute the command ‘make’

Makefiles

- Makefiles can also be used to manage larger projects and carry out other tasks such as clean up.
- When you include .h files that you have written you need to link files together. This can all be done in a Makefile as well.
- The -c flag for g++ will compile the code and create .o files

```
g++ -c HelloWorld.cpp
```

Makefiles

- The code can then be linked by running g++ on the .o files to create an executable.

```
g++ -o HelloWorld HelloWorld.o
```

- This can be put into the Makefile as the following:

```
HelloWorld.o: HelloWorld.cpp
        g++ -c HelloWorld.cpp
HelloWorld: HelloWorld.o
        g++ -o HelloWorld HelloWorld.o
```

Primitive Data Types (ed. by us)

Data Type	Range of Values (Intel x86)
short	-32,768 through 32,787
unsigned short	0 through 65,535
int	-2,147,483,648 through 2,147,483,647
unsigned int, size_t	0 through 4,294,967,295
long	Now usually -2^63 thru 2^63 – 1, i.e. 64-bit
unsigned long	Now usually 0 thru 2^64 – 1 (e.g. on timberlake)
float	Approximately $\pm 10^{-38}$ to 10^{38} with 7 digits of precision
double	Approximately $\pm 10^{-308}$ to 10^{308} with 15 digits of precision
long double[1]	Approximately $\pm 10^{-4932}$ to 10^{4932} with 18 digits of precision (80-bit IEEE extension, maybe longer)
char	The 7-bit ASCII characters
signed char	-128 through 127
unsigned char	0 through 255
wchar_t	The Unicode characters
bool	**true** or **false**

[1] With the Microsoft compiler **long double** is the same as **double**. (True for you VC++ people?)

Numeric Constants

- `1234` is an `int`
- `1234U` or `1234u` is an `unsigned int`
- `1234L` or `1234l` is a `long`
- `1234UL` or `1234ul` is an `unsigned long`
- `1.234` is a `double`
- `1.234F` or `1.234f` is a `float`
- `1.234L` or `1.234l` is a `long double`.

Operator Precedence

Rank	Operator	Operation	Associativity
1	[]	Array subscript	Left
	()	Function call	
	.	Member access	
	->	Member access	
	++ --	Postfix increment or decrement	
2	++ --	Prefix increment or decrement	
	*	Pointer de-referencing operator	
	&	Address of operator	
	+ -	Unary plus or minus	
	!	Complement	
	~	Bitwise complement	
	new	Object creation	
3	* / %	Multiply, divide, remainder	
4	+ -	Addition, Subtraction	
5	<<	Shift left	
	>>	Shift right	

Operator Precedence (2)

Rank	Operator	Operation	Associativity
6	< <=	Less than, Less than or equal	
	> >=	Greater than, Greater than or equal	
7	==	Equal to	
	!	Not equal to	
8	&	Bitwise and	
9	^	Exclusive or	
10	|	Bitwise or	
11	&&	Logical and	
12	||	Logical or	
13	?:	Conditional	
14	=	Assignment	Right
	*= /= &= += -= <<= >>= &= |=	Compound Assignment	

C-Style Casts (ed by us)

- Old style: (*new-type*) *expression*

  This form is inherited from the C programming language and its use is discouraged in C++ programs (but needed? for primitive pointer casts).
- New style: *newType(expression)*. E.g. int(x), Foo(x).
- Newer style: *dynamic_cast<newClass*>(p)* (more later).
- C++ has other type conversion operators (also called cast operators) for conversion among user-defined (i.e. Class) types.
  - These are discussed in lecture, when they are used.

T* v versus T *v

- Using the form

```
double* px;
```

  clearly states that `px` is of type *pointer-to-***`double`**.

- Using the form

```
double *px;
```

  states that the expression `*px` is of type **`double`**, thus `px` must be a *pointer-to-***`double`**.

- Use Foo*, no exceptions!

Multiple Variables in one Declaration

- The declaration:

```
double* px, py;
```

  declares that px is a pointer-to-double, but py is a double.

- To declare multiple pointer variables in one declaration:

```
double *px, *py;
```

- NEVER do either!! (standard style)

The NULL pointer

- The null pointer is a pointer value that points to nothing.
- Internally the value of the null pointer is implementation defined.
- The literal constant 0 is converted to a null pointer.
- Null pointers are converted to **`false`** when used in boolean expressions, and non-null pointers are converted to **`true`**.
- The macro NULL is defined in `<cstddef>` as:

```
#define NULL 0
```

- Future versions of C++ will have a reserved-word for the null pointer literal.

Function Definition (ed. by us)

- Form:

```
return-type  function-name(parameter list)  {
    function body
}
```

- The parameter list is either empty, or a comma-separated list of the form:

```
type-name parameter-name
```

- Function definitions (especially outside namespaces, i.e. “global”) are discouraged, except for operators and some functions associated to a given class, in the same file. See e.g. example files LinkArg.{h,cpp}.

Operator Functions (ed. by us)

- C++ allows class types to be operated on as if they were primitive types.
- You can define operators such as +, -, * etc. to operate.
- Example, if s1 and s2 are strings

```
s1 + s2
```

  represents the string consisting of s1 followed by s2.
- The name of the operator functions is the form `operator@` where @ represents the operator.
- Example:

```
operator+
```

  is the + operator.
- Weirdly, `operator()` allows you to customize the function-application operator. Used like .apply(…) in Java

Arrays and C Strings (ed. by us)

- An array is an object. (In Java. In C++ it's just a pointer. Hence don't use---use **`vector<...>`**.)
- The elements of an array are all of the same type.
- The elements of an array are accessed by an index applied to the subscript operator.

  array-name[*index*]

Declaring an array

- Form:

  type-name array-name`[`*size*`];`

  type-name array-name`[] = {`*initialization list*`};`

- Examples:

```
int scores[5];
string names[] = {"Sally", "Jill", "Hal", "Rick"};
```

- Main difference from Java: you can't put [] next to the type name.

Pointers and Arrays

- C++ performs automatic conversion between array types and pointer types.
- The expression:

  ```
  students[0]
  ```

  and

  ```
  *students
  ```

  are equivalent.
- The expression:

  ```
  a[i]
  ```

  is equivalent to

  ```
  *(a + i)
  ```

  and

  ```
  &a[i]
  ```

  to

  ```
  (a + i)
  ```
- Main importance: this notation carries thru to iterators in the Standard Template Library. Cf. table in text, pp268—269.

Arrays as function arguments

- Arrays are passed to functions as pointers.
- Function parameters may be declared either as pointers or arrays,
  - but the two are equivalent.
- Example:

```
int find(int x[], int n, int target);
int find(int* x, int n, int target);
```

  are equivalent.
- The “int n” is the size of the array, which must be passed separately, since arrays don’t know their bounds. HENCE, pass `vector`s---official advice from Stroustrup himself---and Burn This Slide!

The string class

- The `string` class is defined in the header `<string>`
- Using the string class allows us to manipulate string objects similar to objects of the primitive types.
- Example:

```
string s1, s2;
s1 = "hello";
s2 = s1 + " world";
```

- Text has a comprehensive list of methods.

The <iostream> header

- The header <iostream> declares the following pre-defined streams as global variables:

```
istream cin;  //input from standard input
ostream cout; //output to standard output
ostream cerr; //output to the standard error
```

- Standard input is generally from the keyboard, but may be assigned to be from a file.
- Standard output and standard error are generally to the console, but may be assigned to a file.

The istream class

- The istream class performs input from input streams.
- It defines the extraction operator (>>) for the primitive types and the string class.

Type of operand	Processing
`char`	The first non-space character is read.
`string`	Starting with the first non-space character, characters are read up to the next space.

`int`
`short`
`long`
	If the first non-space character is a digit (or + or -), characters are read until a non-digit is encountered. The sequence of digits is then converted to an integer value of the specified type.

`float`
`double`
`long double`
	If the first non-space character is a digit (or + or -), characters are read as long as they match the syntax of a floating-point literal. The sequence of characters is then converted to a floating-point value of the specified type.

Status Reporting Functions

Member Function	Behavior
`bool eof() const`	Returns **`true`** if there is no more data available from the input stream, and there was an attempt to read past the end.
`bool fail() const`	Returns **`true`** if the input data did not match the expected format, or if there is an unrecoverable error.
`bool bad() const`	Returns **`true`** if there is an unrecoverable error.
`bool operator!() const`	Returns `fail()`. This function allows the `istream` variable to be used directly as a logical variable.
`operator void*() const`	Returns a null pointer if `fail()` is `true`, otherwise returns a non-null pointer. This function allows the use of an `istream` variable as a logical variable.

KWR: Famously fubar, but no one has superseded it.

Reading all input from a stream

```
int n = 0;
int sum = 0;
int i;
while (cin >> i) {
  n++;
  sum += i;
}
if (cin.eof()) {
  cout << "End of file reached\n";
  cout << "You entered " << n << numbers\n";
  cout << "The sum is " << sum << endl;
} else if (cin.bad()) {
  cout << "Unrecoverable i/o error\n";
} else {
  cout << "The last entry was not a valid number\n";
}
```

The ostream class

- The ostream class provides output to an output stream.
- It defines the insertion operator (<<) for primitive types and the string class.

Type of operand	Processing
`char`	The character is output.
`string`	The sequence of characters in the string is output.

`int`
`short`
`long`
	The integer value is converted to decimal and the characters are output. Leading zeros are not output unless the value is zero, in which case a single 0 is output. If the value is negative, the output is preceded by a minus sign.

`float`
`double`
`long double`
	The floating-point value is converted to a decimal representation and output. By default a maximum of six digits is output. If the absolute value is between 10^{-4} and 10^{6}, the output is in fixed format; otherwise it is in scientific format.

Formatting Manipulators in <iostream>

Manipulator	Default	Behavior
noshowpoint	yes	If a floating-point value is a whole number, the decimal point is not shown.
showpoint	no	The decimal point is always shown for floating-point output.
skipws	yes	Sets the format flag so that on input white space (space, newline, or tab) characters are skipped.
noskipws	no	Sets the format flag so that in input white space (space, newline, or tab) characters are read.
right	yes	On output, the value is right-justified.
left	no	On output, the value is left-justified.
dec	yes	The input/output is in base 10.
hex	no	The input/output is in base 16.
fixed	no	Floating-point output is in fixed format
scientific	no	Floating-point output is in scientific format.
ws	no	On input, whitespace is skipped. This is a one-time operation and does not clear the format flag.
endl	no	On output, a newline character is written and the output buffer is flushed.

I/O Manipulators in <iomanip>

Manipulator	Behavior
setw(size_t)	Sets the minimum width of the next output. After this the minimum width is reset to the default value of 0.
setprecision(size_t)	Sets the precision. Depending on the output format, the precision is either the total number of digits (scientific) or the number of fraction digits (fixed). The default is 6.
setfill(char)	Sets the fill character. The default is the space.
resetiosflags(ios_base::fmtflags)	Clears the format flags set in the parameter.
setiosflags(ios_base::fmtflags)	Sets the format flags set in the parameter.

Floating-point output format

- The default floating-point format is called general.
- If you set either fixed or scientific, then to get back to general format you must use the mainiplator call:

```
resetiosflage(ios_base::fixed | ios_base::scientific)
```

Format	Example	Description
Fixed	`123.456789`	Output is of the form `ddd.ffffff` where the number of digits following the decimal point is specified by the precision.
Scientific	`1.2345678e+002`	Output is of the form `d.fffff±ennn` where the number of digits following the decimal point is controlled by the value of precision. (On some systems only two digits for the exponent are displayed.)

General	`1.23456e+006`
	`1234567`
	`123.4567`
	`1.234567e-005`
		A combination of fixed and scientific. If the absolute value is between 10^{-4} and 10^{6}, output is in fixed format; otherwise it is in scientific format. The number of significant digits is controlled by the value of precision.

File Streams

- The header <fstream> defines the classes

`ifstream`	An `istream` associated with a file
`ofstream`	An `ostream` associated with a file

Constructors and the **`open`** function (ed. by us)

Function	Behavior
`ifstream()`	Constructs an `ifstream` that is not associated with a file.
`ifstream(const char* file_name, ios_base::openmode mode = ios_base::in)`	Constructs an `ifstream` that is associated with the named file. By default, the file is opened for input.
`ofstream()`	Constructs an `ofstream` that is not associated with a file.
`ofstream(const char* file_name, ios_base::openmode mode = ios_base::out)`	Constructs an `ofstream` that is associated with the named file. By default, the file is opened for output.
`void open(const char* file_name, ios_base::openmode)`	Associated an `ifstream` or and `ofstream` with the named file and sets the openmode to the specified value.

Use **`ios_base:app`** to append to existing file without zapping it.

Because arg1 is `char*` not `string`, must use `c_str()` method

Openmode Flags

openmode	Meaning
`in`	The file is opened for input.
`out`	The file is opened for output.
`binary`	No translation is made between internal and external character representation.
`trunc`	The existing file is discarded and a new file is written. This is the default and applies only to output.
`app`	Data is appended to the existing file. Applies only to output.

Example File Input (by us)

```
ifstream* infilep;  //lines from “main” in “stringsorts.cpp” (in Java2C++/)
…
if (argc >= 2) {
   infileName = args->at(1);
   infilep = new ifstream(infileName.c_str(), ios_base::in);
   if (! infilep->is_open()) {
      cerr << "Unable to open file: " << infileName << endl;
      return(1);
   }
} else {    //argc == 1, i.e. no arguments given
   cerr << "Usage: stringsorts file n    with n >= 0" << endl;
   return(1);
}
```

Differs from text examples in that the stream is assigned inside a scope. Alas, needs pointers!

Why---(by us, more in lecture)

- To prevent "cloning" streams, C++ "disables" their `operator=` and copy-constructors
  - by declaring them `private`.
- If you try to write "value-based" code like this:

```
ifstream infilep;
if (argc >= 2) {
   infilep = ifstream(...);  //new ifstream would be a "type error"
```

you get the error that they are "private in this context"---and screenfuls more!

- Newer languages are better at telling delayed initialization apart from assignment.

File Pointer Example (cont'd, by us)

```
vector<string> itemsCopy1(n);   //now read up to n strings
int numItems = 0;  //means # of strings stored so far AND next free index
string temp;
while (numItems < n && *infilep >> temp)  {    //test for !fail, see text
   // control here means read was good, so store item
   itemsCopy1[numItems++] = temp;
}
if (infilep->eof()) {
   cout << "Read all " << numItems << " items in " << infileName << endl;
} else if (infilep->bad()) {
   cerr << "Unrecoverable i/o error after "<<numItems<<" items." << endl;
} else {  // stream is still good, so we must have hit n items
   cout << "Read " << numItems << " items from " << infileName << endl;
}
```

Outputting With File Pointers (by us)

```
const string outfileName = infileName + ".out";  //still in stringsorts.cpp
ofstream* outfilep = new ofstream(outfileName.c_str(), ios_base::out);
assert(outfilep->is_open());   //actual code file has different optional test
…
for (int i = 0; i < numItems; i++) {
   (*outfilep) << itemsCopy1[i] << endl;  //puts items on separate lines
}
infilep->close();    //could also do delete(infilep) and delete(outfilep),
outfilep->close();  //which (should!) call close on the files.
```

- For legacy-code reasons, the ifstream and ofstream constructors need a char* old-style string as first arg, hence the calls to the c_str() method.
- Can use ios_base::app as 2nd arg to append not zap outfile.

String Streams

- Defined in the header <sstream>
- Associates an istream or ostream with a string object.

Constructor	Behavior
`explicit istringstream(const string&)`	Constructs an `istringstream` to extract from the given string.
`explicit ostringstream(string&)`	Constructs an `ostringstream` to insert into the given string.
`ostringstream()`	Constructs an `ostringstream` to insert into an internal string.
Member Function	**Result**
`string str() const`	Returns a copy of the string that is the source or destination of the `istringstream` or `ostringstream`.

Using an istringstream

- Assume that the string `person_data` contains:

  `Doe, John 5/15/65`

- We want to split this into `family_name`, `given_name`, `month`, `day`, and `year`.

```
istringstream in(person_data);
in >> family_name >> given_name;
in >> month; // Read the month
in >> c;     // Skip the / character
in >> day;   // Read the day
in >> c;     // Skip the / character
in >> year;  // Read the year
```

Using an ostringstream

- We want to construct the string `person_data` from the component values.

```
ostringstream out;
out << family_name << ", " << given_name << " "
    << month << "/" << day << "/" << year;
string person_data = out.str();
```

String-conversion Convention (by us)

To code a .toString() method, the idiom is:

- Declare an `ostringstream&` variable `out`.
- Output the fields of your object to `out` the way you would with `std::cout`.
- Finally `return out.str();`
- Can shorten the method name to `str()`.
- You still need to write e.g. `cout << obj.str()`, but you'd have to with `operator string()` anyway.
- See LinkArg.{h,cpp} for a simple example.

Strings to Numbers (by us)

- You can read an `int` from the console via `cin` or on any istream `IN` by declaring a variable **`x`** of type **`int`** and doing **`IN >> x;`**
- Same idea for `double` and other types.
- If you've already saved the digits as a string `xstr`, do (with `#include<sstream>`):
  - **`istringstream iss(xstr);`**
  - **`iss >> x;`**
- If `iss` is already declared, do **`iss.str(xstr);`**
- Programs written by us have examples.