Chapter Eight: Streams

Slides by Evan Gallagher

Streams

A ship

Chapter Goals

- To be able to read and write files
- To convert between strings and numbers using string streams
- To process command line arguments
- To understand the concepts of sequential and random access

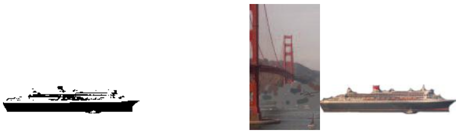
Streams

in the stream

Streams

A very famous bridge
over a “*stream*”

Streams



one at a time

Streams

A ***stream*** of ships

Streams

No more ships in the stream at this time
Let's process what we just input…

Streams

an *input stream* to
that famous city

Reading and Writing Files

- The C++ input/output library is based on the concept of *streams*.
- An *input stream* is a source of data.
- An *output stream* is a destination for data.
- The most common sources and destinations for data are the files on your hard disk.

Streams

Streams

newline characters

```
This is a stream of characters. It could be from
the  keyboard  or  from a file. Each of these is
just a character - even these:  3  -23.73 which,
when input, can be converted to: ints or doubles
or whatever type you like.
(that was a  '\n'  at the end  of the last line)
&*@&^#!%#$ (No, that was –not- a curse!!!!!!!!!!
¥1,0000,0000 (price of a cup of coffee in Tokyo)
Notice that all of this text is very plain -  No
bold or green of italics – just characters – and
whitespace (TABs, NEWLINES and, of course... the
other one you can't see: the space character:
(another '\n')
(&& another)    (more whitespace) and FINALLY:

Aren't you x-STREAM-ly glad this animation is over?
And there were no sound effects!!!
```

Reading and Writing Files

The stream you just saw in action is a plain text file.
No formatting, no colors, no video or music
(or sound effects).

The program can read these sorts of plain text streams of characters from the keyboard, as has been done so far.

Reading and Writing Files

You can also read from files stored on your hard disk:

from plain text files
(as if the typing you saw had been stored in a file)
(or you wanted to write those characters to a disk file).

from a file that has binary information
(a binary file).

The picture of the ship in the stream
of ships is stored as binary information.

Reading and Writing Files

You will learn to read and write both kinds of files.

Reading and Writing Files

To read or write disk files, you use *variables*.

You use *variables* of type:

`ifstream` for input from plain text files.
`ofstream` for output to plain text files.
`fstream` for input and output from binary files.

Opening a Stream

To read anything from a file stream,

you need to *open* the stream.

(The same for writing.)

Opening a Stream

Opening a stream means associating
your stream variable with the disk file.

Opening a Stream

The first step in opening a file is
having the stream variable ready.

Here's the definition of an input
stream variable named `in_file`:

```
ifstream in_file;
```

Looks suspiciously like every other
variable definition you've done
– it is!
Only the type name is new to you.

Opening a Stream

Suppose you want to read data from a file named `input.dat` located in the same directory as the program.

All stream variables are objects so we will use a method.
The `open` method does the trick:

```
in_file.open("input.dat");
```

Opening a Stream

You use the name of the disk file
only when you open the stream.

And the `open` method only accepts C strings.

(More about this later …)

Opening a Stream

File names can contain directory path information, such as:

UNIX

```
  in_file.open("~/nicework/input.dat");
```

Windows

```
  in_file.open("c:\\nicework\input.dat");
```

?

Opening a Stream

File names can contain directory path information, such as:

UNIX

```
  in_file.open("~/nicework/input.dat");
```

Windows

```
  in_file.open("c:\\nicework\input.dat");
```

When you specify the file name as a string literal,
and the name contains backslash characters

(as in a Windows path and filename),
you must supply each backslash *twice*
to avoid having *escape characters* in the string,

like `'\n'`.

Opening a Stream

If you ask a user for the filename, you would
normally use a `string` variable to store their input.

But the `open` method requires a C string.

What to do?

Opening a Stream

Luckily most classes provide the methods we need:
the **`string`** class has the **`c_str`** method
to convert the C++ string to a C string:

```
cout << "Please enter the file name:";
string filename;
cin >> filename;
ifstream in_file;
in_file.open(filename.c_str());
```

Opening a Stream

The **`open`** method is passed C string
version of the filename the user typed:

```
cout << "Please enter the file name:";
string filename;
cin >> filename;
ifstream in_file;
in_file.open(filename.c_str());
```

Closing a Stream

When the program ends,
all streams that you have opened
will be automatically closed.

You *can* manually close a stream with the
`close` member function:

```
in_file.close();
```

Closing a Stream

Manually closing a stream is *only* necessary
if you want to open the file again in
the same program run.

Reading from a Stream

You already know how to read and write using files.

Yes you do:

```
string name;
double value;
in_file >> name >> value;
```

See, I told you so!

`cout`? **`in_file`**?
No difference when it comes to reading using **>>**.

Reading from a Stream

The **>>** operator returns a “not failed” condition,
allowing you to combine an input statement and a test.
A “failed” read yields a **`false`**
and a “not failed” read yields a **`true`**.

```
if (in_file >> name >> value)
{
   // Process input
}
```

Nice!

Reading from a Stream

You can even read ALL the data from a file because running out of things to read causes that same “failed state” test to be returned:

```
while (in_file >> name >> value)
{
   // Process input
}
```

x-STREAM-ly   STREAM-lined
--- Cool!

Writing to a Stream

Here’s everything:

1. *create output stream variable*
2. *open the file*
3. *check for failure to open*
4. *write to file*
5. *congratulate self!*

```
ofstream out_file;
out_file.open("output.txt");
if (in_file.fail()) { return 0; }
out_file << name << " " << value << endl;

out_file << "CONGRATULATIONS!!!" << endl;
```

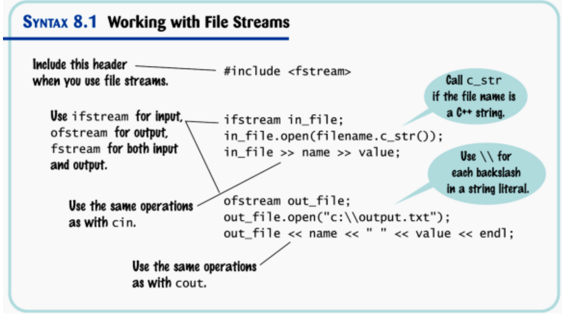
Failing to Open

The `open` method also sets a “not failed” condition.
It is a good idea to test for failure immediately:

```
in_file.open(filename.c_str());
// Check for failure after opening
if (in_file.fail()) { return 0; }
```

Silly user,
bad filename!

Working with File Streams



SYNTAX 8.1 Working with File Streams

Include this header when you use file streams.

```
#include <fstream>

ifstream in_file;
in_file.open(filename.c_str());
in_file >> name >> value;

ofstream out_file;
out_file.open("c:\\output.txt");
out_file << name << " " << value << endl;
```

Use `ifstream` for input, `ofstream` for output, `fstream` for both input and output.

Call `c_str` if the file name is a C++ string.

Use `\\` for each backslash in a string literal.

Use the same operations as with `cin`.

Use the same operations as with `cout`.

Writing to a Stream

Let’s review:

Do you already know everything about writing to files?

But you haven’t started showing writing to files!
How can this be a review already?

But, of course, you already know!

Passing Streams to Functions

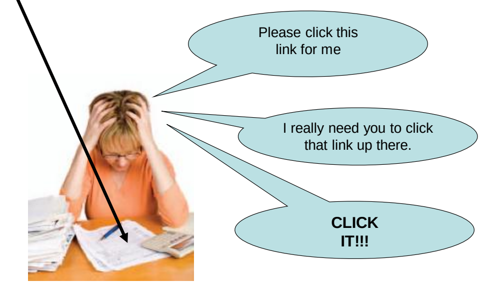
Functions need to be able to process files too,
and there’s an easy rule to follow:

As parameters, streams are
ALWAYS passed by reference.

(Did you notice that “ALWAYS”?)

A Programming Problem

`http://www.ssa.gov/OACT/babynames/`


Please click this link for me

I really need you to click that link up there.

CLICK IT!!!

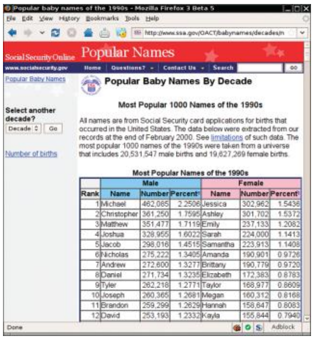
A Programming Problem

Why can I never find Ezmereldza?

A Programming Problem: BABYNAMES

thank you

Now what?



Popular Baby Names By Decade

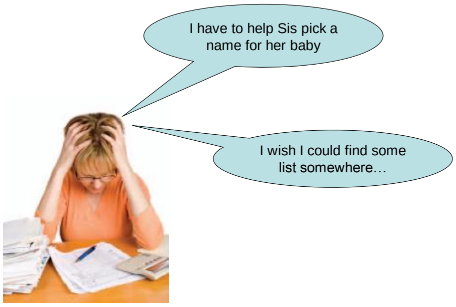
Most Popular 1000 Names of the 1990s

All names are from Social Security card applications for births that occurred in the United States. The data below were extracted from our records at the end of February 2000. See limitations of such data. The most popular 1000 names of the 1990s were taken from a universe that includes 20,531,547 male births and 19,627,269 female births.

Most Popular Names of the 1990s

	Male			Female		
Rank	Name	Number	Percent	Name	Number	Percent
1	Michael	462,085	2.2506	Jessica	302,962	1.5436
2	Christopher	361,250	1.7595	Ashley	301,702	1.5372
3	Matthew	351,477	1.7119	Emily	237,133	1.2082
4	Joshua	328,955	1.6022	Sarah	224,000	1.1413
5	Jacob	298,016	1.4515	Samantha	223,913	1.1408
6	Nicholas	275,222	1.3405	Amanda	190,901	0.9726
7	Andrew	272,600	1.3277	Brittany	190,779	0.9720
8	Daniel	271,734	1.3235	Elizabeth	172,383	0.8783
9	Tyler	262,218	1.2771	Taylor	168,977	0.8609
10	Joseph	260,365	1.2681	Megan	160,312	0.8168
11	Brandon	259,299	1.2629	Hannah	158,647	0.8083
12	David	253,193	1.2332	Kayla	155,844	0.7940

A Programming Problem


I have to help Sis pick a name for her baby

I wish I could find some list somewhere…

A Programming Problem: Baby Names

Let’s write a program that might help Ezmereldza help her sister.

thank you

A Programming Problem: Baby Names

After copying the data from the Social
Security Administration’s table to a text file,
we analyze the format of the file.

A Programming Problem: Baby Names

Each line in the file contains seven entries:

- The rank (from 1 to 1,000)
- The name, frequency, and percentage of the male name of that rank
- The name, frequency, and percentage of the female name of that rank

An example line from the file:

```
10 Joseph 260365 1.2681 Megan 160312 0.8
```

A Programming Problem: Baby Names

We will display the names of the top 50%
of the names stored in our file of names.

A Programming Problem: Baby Names

To process each line, we first read the rank:

```
int rank;
in_file >> rank;
```

We then read a set of three values for that boy’s name:

```
string name;
int count;
double percent;
in_file >> name >> count >> percent;
```

Then we repeat that step for a girl’s name.

A Programming Problem: Baby Names

Repeating a process reminds us of a good design principle:

Write a function to do this:

```
string name;
int count;
double percent;
in_file >> name >> count >> percent;
```

A Programming Problem: Baby Names

And something else,

“ALL’S GOOD THAT ENDS WELL?”
(No.)

“ALL GOOD BOYS DO FINE?”
(No, that was from a music lesson.)

“ALL MEANS NECESSARY?”
(What?)

Aha! It was:

“ALWAYS pass streams by reference”

A Programming Problem: Baby Names

To stop processing after reaching 50%, we can add up the frequencies we read and stop when they reach 50%.

However, it turns out to be a bit simpler to have “total” variables for boys and girls and initialize these with 50.

Then we’ll subtract the frequencies as we read them.

A Programming Problem: Baby Names

ch08/babynames.cpp

```
/**
Reads name information, prints the name if total >= 0, and
adjusts the total.
@param in_file the input stream
@param total the total percentage that should still be
processed
*/
void process_name(ifstream& in_file, double& total)
{
   string name;
   int count;
   double percent;
   in_file >> name >> count >> percent;
   // Check for failure after each input
   if (in_file.fail()) { return; }
   if (total > 0) { cout << name << " "; }
   total = total - percent;
}
```

A Programming Problem: Baby Names

When the total for boys falls below 0, we stop printing boys’ names.

Same for girls’ names
– which means this can be part of the function.

When both totals fall below 0, we stop reading.

A Programming Problem: Baby Names

ch08/babynames.cpp

```
int main()
{
   ifstream in_file;
   in_file.open("babynames.txt");
   // Check for failure after opening
   if (in_file.fail()) { return 0; }
   double boy_total = 50;
   double girl_total = 50;
   while (boy_total > 0 || girl_total > 0)
   {
      int rank;
      in_file >> rank;
      if (in_file.fail()) { return 0; }
      cout << rank << " ";
      process_name(in_file, boy_total);
      process_name(in_file, girl_total);
      cout << endl;
   }
  return 0;
}
```

A Programming Problem: Baby Names

You’ll get this when you read the code:

ch08/babynames.cpp

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
```

Reading Text Input

There are more ways to read from stream variables than you have seen before,
and there are some details you need to understand.

These follow…

Reading `strings`

What really happens when reading a **`string`**?

```
string word;
in_file >> word;
```

1. If any reading can be done at all, all whitespace is skipped (whitespace is this set: **`'\t' '\n' ' '`**).
2. The first character that is not white space is added to the string **`word`**. More characters are added until either another white space character occurs, or the end of the file has been reached.

C++ for Everyone by Cay Horstmann

Not Wanting What You Get

This alfabetiple sgebbie has a B in it and I don't like Bees!
Take it back.

Dorathea!

Please.

C++ for Everyone by Cay Horstmann

Reading Characters

It is possible to read a single character
– including whitespace characters.

```
char ch;
in_file.get(ch);
```

The **`get`** method also returns the "not failed" condition so:

```
while (in_file.get(ch))
{
   // Process the character ch
}
```

C++ for Everyone by Cay Horstmann

Not Wanting What You `.get()`

That which was got
can be ungot!

```
in_file.unget();
```

Dorathea!

Thank you.

C++ for Everyone by Cay Horstmann

Reading the Whole File Character by Character

The **`get`** method makes it possible to process
a whole file one character at a time:

```
char ch;
while (in_file.get(ch))
{
    // Process the character ch
}
```

C++ for Everyone by Cay Horstmann

One-Character Lookahead

You can look at a character after reading it
and then put it back if you so decide.

However you can only put back
the *very last* character that was read.

This is called *one-character lookahead*.

C++ for Everyone by Cay Horstmann

Reading a Number Only If It Is a Number

A typical situation for lookahead is to look for numbers before reading them so that a failed read won't happen:

```
char ch;
in_file.get(ch);
if (isdigit(ch)) // Is this a number?
{
   // Put the digit back so that it will
   // be part of the number we read
   in_file.unget();

   // Read integer starting with ch
   int n;
   data >> n;
}
```

C++ for Everyone by Cay Horstmann

Character Functions in `<cctype>`

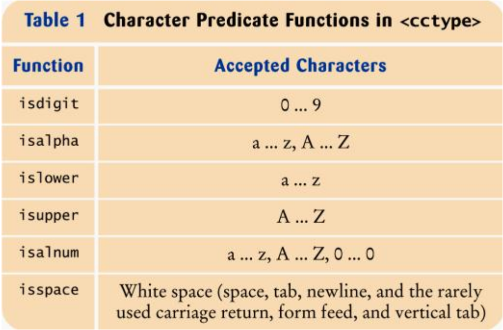
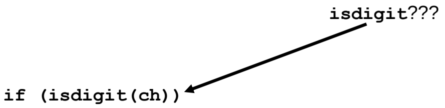


Table 1 Character Predicate Functions in `<cctype>`

Function	Accepted Characters
`isdigit`	0 ... 9
`isalpha`	a ... z, A ... Z
`islower`	a ... z
`isupper`	A ... Z
`isalnum`	a ... z, A ... Z, 0 ... 0
`isspace`	White space (space, tab, newline, and the rarely used carriage return, form feed, and vertical tab)

C++ for Everyone by Cay Horstmann

Reading a Number Only If It Is a Number



`isdigit`???

`if (isdigit(ch))`

C++ for Everyone by Cay Horstmann

Reading A Whole Line: `getline`

The function (it's not a method):

`getline()`

is used to read a whole line
up to the next newline character.

C++ for Everyone by Cay Horstmann

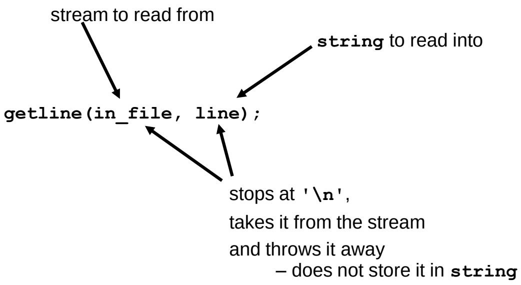
Character Functions in `<cctype>`

The **`isdigit`** function is one of several functions that deal with characters.

`#include <cctype>` is required.

C++ for Everyone by Cay Horstmann

Reading A Whole Line: `getline`



stream to read from

`string` to read into

`getline(in_file, line);`

stops at `'\n'`,
takes it from the stream
and throws it away
– does not store it in **`string`**

C++ for Everyone by Cay Horstmann

Reading A Whole Line: `getline`

Until the next newline character,
all whitespace and all other characters
are taken and stored into the string
– except the newline character – which is just “thrown away”

The *only* type that can be read into is the **`string`** type.

```
string line;
getline(in_file, line); // stops at '\n'
```

Reading A Whole Line: `getline`

The **`getline`** function, like the others we’ve seen, returns the “not failed” condition.

To process a whole file line by line:

```
string line;
while( getline(in_file, line))
{
   // Process line
}
```

Processing a File Line by Line

Reading one line and then processing that line,
taking it apart to get the individual pieces of data from it,
is a typical way of working with files.

Processing a File Line by Line

Here is a top secret online CIA file:

`http://www.cia.gov/library/publications/the-world-factbook/`

Don’t tell anyone, but it looks like this:

```
China 1330044605
India 1147995898
United States 303824646
...
```

Each line has: country name, its population, a newline character.

(And there is some whitespace in there too.)

Processing a File Line by Line

After having copied the secret contents from the website into a text file we will read this file line by line:

```
// we've opened the file
// but we can't tell you it’s name or...

string line;
while( getline(in_file, line))
{
   // Process line
}
```

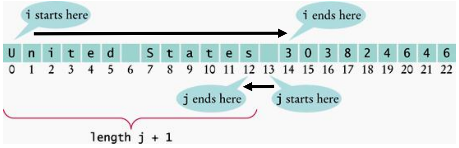
Processing a File Line by Line

To extract the data from the **`line`** variable, you need to find out where the name ends and the number starts**.**

Processing a File Line by Line

```
// Locate the first digit of population
int i = 0;
while (!isdigit(line[i])) { i++; }

// Find end of country name
int j = i - 1;
while (isspace(line[j])) { j--; }
```



U	n	i	t	e	d		S	t	a	t	e	s		3	0	3	8	2	4	6	4	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Processing a File Line by Line

All clear – make the extraction at 09:01 hours.

(that's CIA-speak for: use **string** methods to "extract" the country name and population values)

(we guess "at 09:01" hours means now)

```
string country_name = line.substr(0, j + 1);
string population = line.substr(i);
```

Processing a File Line by Line

Has the CIA made a blunder?
Aren't these being stored as **string**s?

(Is that a security risk?)

```
string country_name = line.substr(0, j + 1);
string population = line.substr(i);
```

Processing a File Line by Line

No, the CIA is quite capable.

Later (09:01 hours),
we will see their "*nice*" (secret) method
for "*extracting*" information from
a (captive) undercover numeric operative
using **std::string** as an alias:

the stringstream
CONTINGENCY

TO BE
CONTINUED…

Writing Text Output

Recall that you know how to output to a file.

(OK)

Writing Text Output

You use the operator >> to send
strings and numbers to an output stream:

```
ofstream out_file;
out_file.open("output.txt");
if (in_file.fail()) { return 0; }
out_file << name << " " << value << endl;
```

Writing Text Output

But what about a single character?

The `put` method:

```
out_file.put(ch);
```

Formatting Output – Manipulators

I don't hear anything.

Not now, but when I ***manipulate*** that red slider,
you'd better have your earplugs ready!

OK, there' in. Go ahead

Nice. What's the name of this tune? *"Sliders On A Slide"*

We're gonna use it in a pretty aggressive, *manipulative* ad.

Formatting Output – Manipulators

Formatting Output – Manipulators

To control how the output is *formatted*,
you use *stream manipulators*.

A *manipulator* is an object that:

- is sent to a stream using the >> operator.
- affects the behavior of the stream.

Formatting Output – Manipulators

I don't hear anything.

Not now, but when I ***manipulate*** that red slider,
you'd better have your earplugs ready!

OK, there' in. Go ahead…

Formatting Output – Manipulators

You already know one:

`endl`

Yes, `endl` is a manipulator.

Formatting Output – Manipulators

When you send the **endl** manipulator to the screen:

```
cout << endl;
```

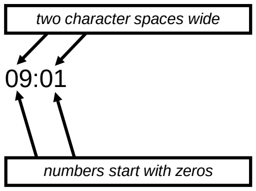
endl causes the cursor
to go to the beginning of the next line.

Moving the cursor is definitely an *affect*ation!

Wait…

Formatting Output – Manipulators

Recall that CIA time display?



Formatting Output – Manipulators

```
cout << endl;
```

Sorry, make that:

“mo *ffect*
of sendi to **cout**,

(sending t stream)
(causing **the** utput stream)

Enough of the
ENGLISH
lesson!

Formatting Output – Manipulators

Recall that CIA time display?

09:01

very
military!

Formatting Output – Manipulators

It’s about *sending the manipulator*:

```
output_stream << manipulator
```

Some other output manipulators:

Formatting Output – Manipulators

Use **setfill** when you need to pad
numbers with leading zeroes.

```
strm << setfill('0')
   << setw(2) << hours << ":"
   << setw(2) << minutes
   << setfill(' ');
```

Formatting Output – Manipulators

Use **setfill** when you need to pad numbers with leading zeroes.
To set the width in which to display, use **setw**.

```
strm << setfill('0')
   << setw(2) << hours << ":"
   << setw(2) << minutes
   << setfill(' ');
```

C++ for Everyone by Cay Horstmann

Formatting Output – A Test

See if you can write the output statements to accomplish the following outputs to the stream variable, **strm**, which is already opened.

You can ask the person with the clicker or pointer, or whoever is in charge of this presentation, to go back to the slide with the table of manipulators.

Be sure to say please…

C++ for Everyone by Cay Horstmann

Formatting Output – Manipulators

Use **setfill** when you need to pad numbers with leading zeroes.
To set the width in which to display, use **setw**.

```
strm << setfill('0')
   << setw(2) << hours << ":"
   << setw(2) << minutes
   << setfill(' ');
```

That last **setfill** re-sets the fill back to the default space character.

C++ for Everyone by Cay Horstmann

Formatting Output – A Test

Produce this output:

```
12.3457 1.23457e+08
```

The code:

```
strm << 12.3456789 << " " << 123456789.0;
```

(will this be on the test?)

C++ for Everyone by Cay Horstmann

Formatting Output – Manipulators

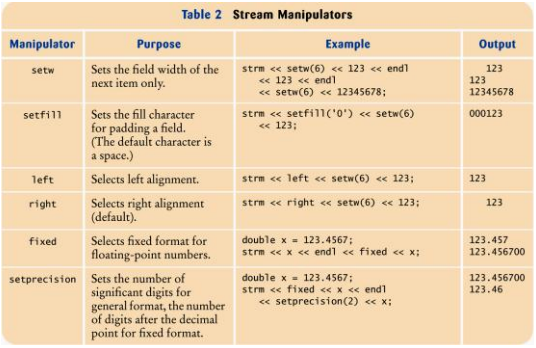


Table 2 Stream Manipulators

Manipulator	Purpose	Example	Output

setw	Sets the field width of the next item only.	strm << setw(6) << 123 << endl
		  << 123 << endl
		  << setw(6) << 12345678;
			123
			123
			12345678

setfill	Sets the fill character for padding a field. (The default character is a space.)	strm << setfill('0') << setw(6) << 123;	000123
left	Selects left alignment.	strm << left << setw(6) << 123;	123
right	Selects right alignment (default).	strm << right << setw(6) << 123;	123

fixed	Selects fixed format for floating-point numbers.	double x = 123.4567;
		strm << x << endl << fixed << x;
			123.457
			123.456700

setprecision	Sets the number of significant digits for general format, the number of digits after the decimal point for fixed format.	double x = 123.4567;
		strm << fixed << x << endl
		  << setprecision(2) << x;
			123.456700
			123.46

C++ for Everyone by Cay Horstmann

Formatting Output – A Test

Produce this output:

```
12.3457 1.23457e+08
```

The code:

```
strm << 12.3456789 << " " << 123456789.0;
```

OK, that was sort of a trick question.
We did nothing!
The default precision and notation is call the *general* format.
Some decimal fractions are shown in *scientific notation*, and the *default* precision to round off to is used.

C++ for Everyone by Cay Horstmann

Formatting Output – A Test

The `fixed` manipulator was in the table, but `scientific` wasn’t shown (it also is a manipulator).

The question was about the *default* settings.

The lesson to be learned is:

Always take charge of formatting your output

Formatting Output – A Test

Produce this output:

```
12.345679 123456789.000000
```

The code:

```
strm << setprecision(6) << fixed
   << 12.3456789 << " " << 123456789.0;
```

Good

Formatting Output – A Test

Produce this output (note that the column width is 10):

```
       123
      4567
```

The code:

```
strm << setw(10) << 123 << endl
   << setw(10) << 4567;
```

Excelle
nt

Formatting Output – A Test

Produce this output (note there are 10 dashes on each side):

```
----------|----------
Count:            177
----------|----------
```

The code:

```
strm << "----------|----------\n"
   << left << setw(10) << "Count:"
   << right << setw(11) << 177
   << "----------|----------\n";
```

An A++ in C++!

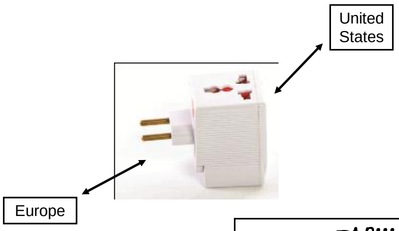
Welcome to CIA headquarters

The time is now:

09:01

It’s stringstream time

Adapters



ZAP!!!
you are transformed

Stream Adapters

In order to “extract” numeric information held in a **`string`**, it would be nice if we could *adapt* the **`string`** type to have the same interface as stream types

– then we could use **>>** and **<<** on **`string`**s.

`istringstream` and `ostringstream`

The **`istringstream`** and **`ostringstream`** are adapters that do just that!

The **`<sstream>`** header is required.

Normal Input Stream – `istream`, `ifstream`

Suppose the user is told to input a date in this form:

```
January 24, 1973
```

The code would be:

```
string month;
int day;
string comma;
int year;
cin >> month >> day >> comma >> year;
```

Normal.

The `istringstream` Type

What if that **`"January 24, 1973"`** were in a **`string`**?

If we had an **`istringstream`** variable named **`strm`**, the only difference would be where to read from:

```
istringstream strm;
string month;
int day;
string comma;
int year;
strm >> month >> day >> comma >> year;
```

No longer normal – but very nice!

The `istringstream` Type

To put **`"January 24, 1973"`** into a **`string`** you use the **`str`** method:

```
istringstream strm;
strm.str("January 24, 1973");
string month;
int day;
string comma;
int year;
strm >> month >> day >> comma >> year;
```

Not really initialization, more like assignment – but neither!

The `istringstream` Type

A function that converts a **`string`** to an integer would be nice:

```
int string_to_int(string s)
{
   istringstream strm;
   strm.str(s);
   int n = 0;
   strm >> n;
   return n;
}
```

This CIA-style extraction is becoming easy!
(That’s good, right?)

istringstream and ostringstream

What's the opposite of extraction?

Insertion.

The "real" name of >> is *extraction*
and << is *insertion*…

…insert into and extract from streams.

The ostringstream Type

An **ostringstream** variable can be
used to "store" string and numbers in a **string**.
The **str** method is used to "extract" the a whole **string**.
The numbers can be formatted as before, and
the output operator works the same as before.

```
string month = "January";
int day = 24;
int year = 1973;
ostringstream strm;
strm << month << " " << day << "," << year
   << " - "
   << fixed << setprecision(5) << 10.0 / 3;
string output = strm.str();
```

The ostringstream Type

A function for numbers to **string**s:

```
string int_to_string(int n)
{
   ostringstream strm;
   strm << n;
   return strm.str();
}
```

Command Line Arguments

Depending on the operating system and C++ development system used, there are different methods of starting a program:

– Select "Run" in the compilation environment.

– Click on an icon.

– Type the program name at a prompt in a command shell window (called "invoking the program from the command line").

Command Line Arguments

In each of these methods, the person starting the program
might want to pass some information in – to where?

To the **main** function!

*Someone calls the **main** function?*

Command Line Arguments

This is how a command shell window might look
where the user is starting the program named **prog**
by typing this command:

```
prog -v input.dat
```

prog is the program name (your C++ program).

Command Line Arguments

This is how a command shell window might look
where the user is starting the program named `prog`
by typing this command:

```
prog -v input.dat
```

`prog` is the program name (your C++ program).
`-v` and `input.dat` are command line arguments

The `-` in `-v` typically indicates an option.

Command Line Arguments

The second parameter is an array of C strings
that hold the command line arguments themselves.

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

`argv` for argument vector
(not a “real” vector)

Command Line Arguments

`main` must be set up differently to be
ready for command line arguments:

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

Command Line Arguments

Given that the user typed:

`argv`

0	1	2
prog	-v	input.dat

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

`argc` is 3
`argv` contains these three C strings:
`argv[0]: "prog"`
`argv[1]: "-v"`
`argv[2]: "input.dat"`

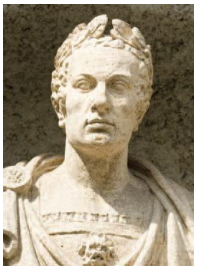
Command Line Arguments

The first parameter is the count of the
number of strings on the command line,
including the name of the command (program).

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

`argc` for argument count

Programming: Encrypting A File



The famous Roman emperor Mxolxv#Fhdvdu
(name encrypted for security reasons)
(on orders issued at 09:02 from the CIA)

Programming: Encrypting A File

Mxolxv#Fhdvdu invented an *encryption scheme*, or *cipher*, to help him keep his military *(and other)* communications secret – that is: undecipherable.

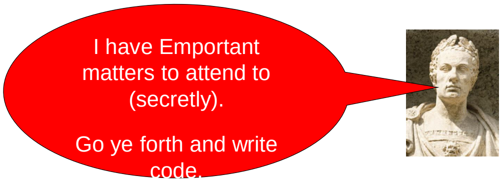
(Actually he didn't invent it, but no one in his right mind would say that to his face)

Programming: Encrypting A File

Hear ye, Hear ye:

Whilst ignoring 2000 years of progress in cryptology
And
by Emperonic decree:

Be ye ordered to encode this *splendid* cipher.


I have Emportant matters to attend to (secretly).

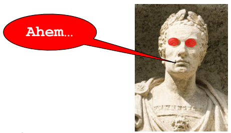
Go ye forth and write code.

Programming: Encrypting A File

These days, the *Fhdvdu Cipher*, as it is now known, is not considered a "strong" encryption algorithm.

It's not even a sort-of-strong algorithm.

It really just a pathetic attempt at an encryption algorithm



but he's the emperor and we are not.

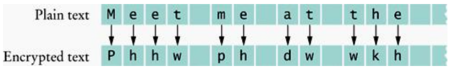
Programming: Encrypting A File

We will use the emperor's name for our executable program and there will be three command line arguments:

1. An optional `-d` flag to indicate decryption instead of encryption
2. The input file name
3. The output file name

Programming: Encrypting A File

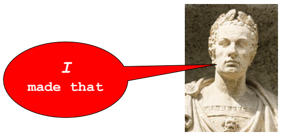
The *Fhdvdu Cipher* :



Plain text	M	e	e	t		m	e		a	t		t	h	e
Encrypted text	P	h	h	w		p	h		d	w		w	k	h

Take each character and move over three characters.

Much better than Pig-Latin!



Programming: Encrypting A File

Sample command lines:

To encrypt the file `input.txt` and place the result into `encrypt.txt`:

```
fhdvdu input.txt encrypt.txt
```

To decrypt the file `encrypt.txt` place the result into `output.txt`:

```
fhdvdu -d encrypt.txt output.txt
```

Programming: Encrypting A File

ch08/*fhdvdu*.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
using namespace std;

/**
   Encrypts a stream using the Fhdvdu cipher.
   @param in the stream to read from
   @param out the stream to write to
   @param k the encryption key
*/
void encrypt_file(ifstream& in, ofstream& out, int k)
{
   char ch;
   while (in.get(ch))
   {
      out.put(ch + k);
   }
}
```

The Fhdvdu Cipher

C++ for Everyone by Cay Horstmann

Programming: Encrypting A File

ch08/*fhdvdu*.cpp

```
         else if (file_count == 2) // The second file name
         {
            out_file.open(arg.c_str());
            if (out_file.fail())
            {
               cout << "Error opening output file "
                  << arg << endl;
               return 1;
            }
         }
      }
   }
```

C++ for Everyone by Cay Horstmann

Programming: Encrypting A File

ch08/*fhdvdu*.cpp

```
int main(int argc, char* argv[])
{
   int key = 3;
   int file_count = 0; // The number of files specified
   ifstream in_file;
   ofstream out_file;

   // Process all command-line arguments
   for (int i = 1; i < argc; i++)
   {
      // The currently processed argument
      string arg = argv[i];
      if (arg == "-d") // The decryption option
      {
         key = -3; }
```

C++ for Everyone by Cay Horstmann

Programming: Encrypting A File

ch08/*fhdvdu*.cpp

```
   // Exit if the user didn't specify two files
   if (file_count != 2)
   {
      cout << "Usage: "
         << argv[0] << " [-d] infile outfile" << endl;
      return 1;
   }

   encrypt_file(in_file, out_file, key);
   return 0;
}
```

C++ for Everyone by Cay Horstmann

Programming: Encrypting A File

ch08/*fhdvdu*.cpp

```
      else // It is a file name
      {
         file_count++;
         if (file_count == 1) // The first file name
         {
            in_file.open(arg.c_str());
            // Exit the program if opening failed
            if (in_file.fail())
            {
               cout << "Error opening input file "
                  << arg << endl;
               return 1;
            }
         }
```

C++ for Everyone by Cay Horstmann

Programming: Encrypting A File

+\hv/#zh#xvhg#wklv#surjudp#wr#hqfu|sw#wkh#qdph=#Mxolxv#Fhdvdu,
+Lw*v#fdoohg#wkh#Fhdvdu#Flskhu,

(Yes, we used this program to encrypt Julius Caesar)
(It's called the Caesar Cipher)

C++ for Everyone by Cay Horstmann

Programming: Encrypting A File

It’s the **Caesar Cipher**.

We said it again to make sure you don’t go out and tell someone you learned about the fhdvdu cipher from us!

Random Access

It doesn’t really mean random as in **`srand`** and **`rand`** and that sort of processing!

Random means that you can read and modify any item stored at any location in the file – directly going to it.

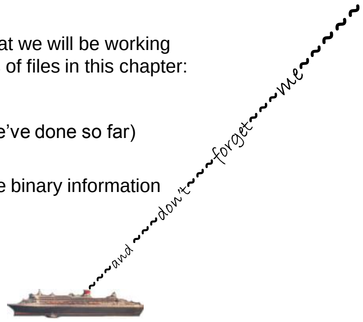
To get to the 42nd item in the file you don’t have to read all 41 items before it.

Random Access and Binary Files

Remember that we will be working with two kinds of files in this chapter:

Plain text files
(everything we’ve done so far)

Files that have binary information
(a binary file)



Random Access

Random access means:

No standing in line.

And you can start
with the cake!

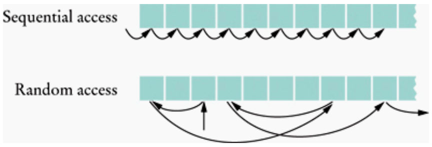
Sequential Access and Random Access

There also two methods of working with files:

Sequential Access
as we’ve been doing
– one input at a time starting at the beginning

Random Access
Random?
`rand`?

Sequential Access and Random Access



Random Access

Pretty obviously `cin` and `cout` aren't random access.
Only file streams are.

You mean this computer isn't smart enough to jump to the word I'm going to type three minutes from now?

What is this computer good for?

Random Access

To change the get positions, you use this method:

```
strm.seekg(position);
```

`seekg` means move the get position
"g" as in "get" – get it?

The parameter value is how many *bytes* from the beginning of the file to move the get position to.

Random Access

The screen has a cursor so that the user knows where she is typing.

Files have two special positions:

the *put* position – where the next write will go.

the *get* position – where the next read will be.

Random Access

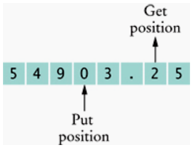
`seekp` does the same for the put position

```
strm.seekp(position);
```

Random Access

The 0 will be overwritten next.

The 2 will be read next.



Random Access

You can also find out where these positions currently are:

```
g_position = strm.tellg();
p_position = strm.tellp();
```

Binary Files

Many files, in particular those containing
images and sounds, do not store information
as text but as binary numbers.

The meanings and positions of these binary
numbers must be known to process a binary file.

Binary Files

Data is stored in files as sequences of bytes,
just as they are in the memory of the computer.

(Each byte has a value between 0 and 255.)

To store the word “CAB” takes four bytes:
`67 65 66 00`

The binary data in an image has a special
representation as a sequence of bytes
– but it’s still just a bunch of numbers.

Binary Files

Binary files have different ways of
opening and for reading and writing.

Opening Binary Files

To open a binary file for reading and writing,
use this version of the **`open`** method:

```
fstream strm;
strm.open("img.gif", ios::in | ios::out | ios::binary);
```

That’s the “vertical bar” – the capital backslash.

`ios::in` and **`ios::out`** allow us to
read from and write into the same file.
For plain files we could do only one or the other.

The **`ios::binary`** means, well, um… it’s a binary file.

Opening Binary Files

To read from a binary file you cannot use the **>>** operator
Use the **`get`** method:

```
int input = strm.get();
```

In a binary file stream, this reads one byte as a number,
a value between 0 and 255.

Opening Binary Files

A “real” **`int`**, like **`1822327`**,
takes *four* bytes to store on most systems.

To read a “real” **`int`**, you will have to do *four* reads

– and some arithmetic.

(Or write a function to do this!)

Processing Image Files

To process image files, or any binary file,
you must know how everything is arranged in the file.

Processing Image Files

The BMP image file format is pretty simple.
So we will use it in the following program.

In fact, we'll the use the most simple of the
several versions of the BMP format:

the 24-bit true color format

Processing Image Files

A file format doesn't mean things like `precision` or `left`.

It means things like:

- What *thing* does this byte represent?
- How many bytes does it take to represent this *thing*?

There are lots of "things" you must
know before opening an image file
(or any binary file).

Processing Image Files: The BMP File Format

The BMP file format for 24-bit true color format:

Each pixel's (picture element) color is represented
in RGB form – Red, Green, and Blue amounts.

In the file, each pixel is represented as
a sequence of three bytes:

- a byte for the blue value (B)
- a byte for the green amount (G)
- a byte for the red amount (R)

Processing Image Files: The BMP File Format

Here are some RGB values stored in a BMP file
(you'll notice that it's really stored as BGR):

Cyan (a mixture of blue and green) is the bytes: `255 255 0`

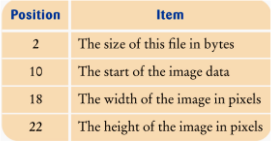
Pure **red** is the values: `0 0 255` (no blue, no green, all red)

Medium gray is `128 128 128` (half of 255 for all three)

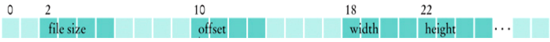
Processing Image Files: The BMP Header

Most files start with some information about
the contents called the *header*.

A BMP file is no different:



Position	Item
2	The size of this file in bytes
10	The start of the image data
18	The width of the image in pixels
22	The height of the image in pixels

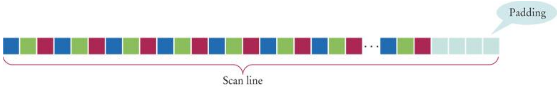


Processing Image Files: The BMP File Format

The image itself is represented as a sequence
of pixel rows (a scan line),
starting with the bottom row in the image.

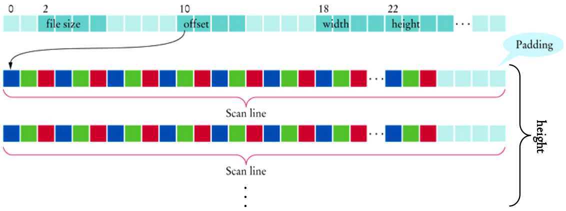
Each pixel row contains a sequence of BGR bytes.

The end of the row is padded with additional bytes so
that the number of bytes in the row is divisible by 4.



Processing Image Files: The BMP File Format

There would be
height scans lines
each *width * 3* bytes long
(rounded up to a multiple of 4)

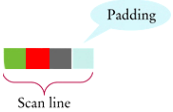


Processing Image Files: The BMP File Format

For example,
if a row consisted of merely three pixels,
one cyan, one red, one medium gray,
there would three padding bytes.

The numbers would be:

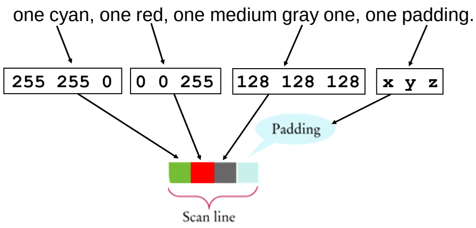
```
255 255 0   0 0 255   128 128 128   x y z
```



Processing Image Files: The BMP File Format

Now that you know all there is to know about BMP files
for 24-bit true color images, we'll write code to create the
negative of an input image file:

Processing Image Files: The BMP File Format



one cyan, one red, one medium gray one, one padding.

```
255 255 0 | 0 0 255 | 128 128 128 | x y z
```

Processing Image Files: The BMP File Format

We will create the negative of each pixel by
subtracting the R, G, and B values from 255.

Of course that will be a function!

Program to Produce the Negative of a BMP Image File

ch08/imagemod.cpp

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
/**
 Processes a pixel by forming the negative.
 @param blue the blue value of the pixel
 @param green the green value of the pixel
 @param red the red value of the pixel
*/
void process(int& blue, int& green, int& red)
{
   blue = 255 - blue;
   green = 255 - green;
   red = 255 - red;
}
```

Program to Produce the Negative of a BMP Image File

ch08/imagemod.cpp

```
/**
 Gets an integer from a binary stream.
 @param stream the stream
 @param offset the offset at which to read the integer
 @return the integer starting at the given offset
*/
int get_int(fstream& stream, int offset)
{
   stream.seekg(offset);
   int result = 0;
   int base = 1;
   for (int i = 0; i < 4; i++)
   {
      result = result + stream.get() * base;
      base = base * 256;
   }
   return result;
}
```

Program to Produce the Negative of a BMP Image File

ch08/imagemod.cpp

```
int main()
{
   cout << "Please enter the file name: ";
   string filename;
   cin >> filename;
   fstream stream;

   // Open as a binary file
   stream.open(filename.c_str(),
      ios::in|ios::out|ios::binary);

   // Get the image dimensions
   int file_size = get_int(stream, 2);
   int start = get_int(stream, 10);
   int width = get_int(stream, 18);
   int height = get_int(stream, 22);
```

Program to Produce the Negative of a BMP Image File

ch08/imagemod.cpp

```
   // Scan lines must occupy multiples of four bytes
   int scanline_size = width * 3;
   int padding = 0;
   if (scanline_size % 4 != 0)
   {
      padding = 4 - scanline_size % 4;
   }
   if (file_size != start +
      (scanline_size + padding) * height)
   {
      cout << "Not a 24-bit true color image file.‟
         << endl;
      return 1;
   }
```

Program to Produce the Negative of a BMP Image File

ch08/imagemod.cpp

```
   // Go to the start of the pixels
   stream.seekg(start);

   // For each scan line
   for (int i = 0; i < height; i++)
   {
      // For each pixel
      for (int j = 0; j < width; j++)
      {
         // Go to the start of the pixel
         int pos = stream.tellg();

         // Read the pixel
         int blue = stream.get();
         int green = stream.get();
         int red = stream.get();
```

Program to Produce the Negative of a BMP Image File

ch08/imagemod.cpp

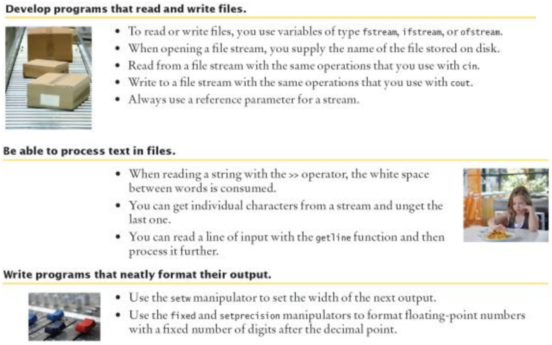
```
         // Process the pixel
         process(blue, green, red);

         // Go back to the start of the pixel
         stream.seekp(pos);

         // Write the pixel
         stream.put(blue);
         stream.put(green);
         stream.put(red);
      }

      // Skip the padding
      stream.seekg(padding, ios::cur);
   }
   return 0;
}
```

CHAPTER SUMMARY



Develop programs that read and write files.

- To read or write files, you use variables of type `fstream`, `ifstream`, or `ofstream`.
- When opening a file stream, you supply the name of the file stored on disk.
- Read from a file stream with the same operations that you use with `cin`.
- Write to a file stream with the same operations that you use with `cout`.
- Always use a reference parameter for a stream.

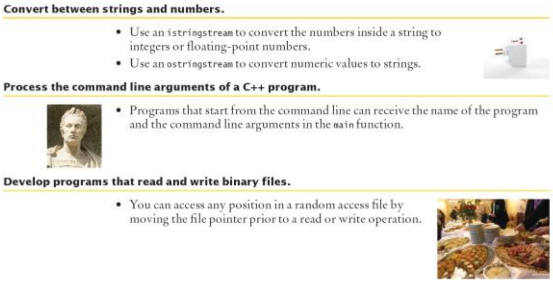
Be able to process text in files.

- When reading a string with the >> operator, the white space between words is consumed.
- You can get individual characters from a stream and unget the last one.
- You can read a line of input with the `getline` function and then process it further.

Write programs that neatly format their output.

- Use the `setw` manipulator to set the width of the next output.
- Use the `fixed` and `setprecision` manipulators to format floating-point numbers with a fixed number of digits after the decimal point.

CHAPTER SUMMARY



Convert between strings and numbers.

- Use an `istringstream` to convert the numbers inside a string to integers or floating-point numbers.
- Use an `ostringstream` to convert numeric values to strings.

Process the command line arguments of a C++ program.

- Programs that start from the command line can receive the name of the program and the command line arguments in the `main` function.

Develop programs that read and write binary files.

- You can access any position in a random access file by moving the file pointer prior to a read or write operation.

End Chapter Eight

Slides by Evan Gallagher