



Chapter Eight: Streams, Part II

Chapter Goals

- To convert between strings and numbers using string streams
- To process command line arguments
- To understand the concepts of sequential and random access

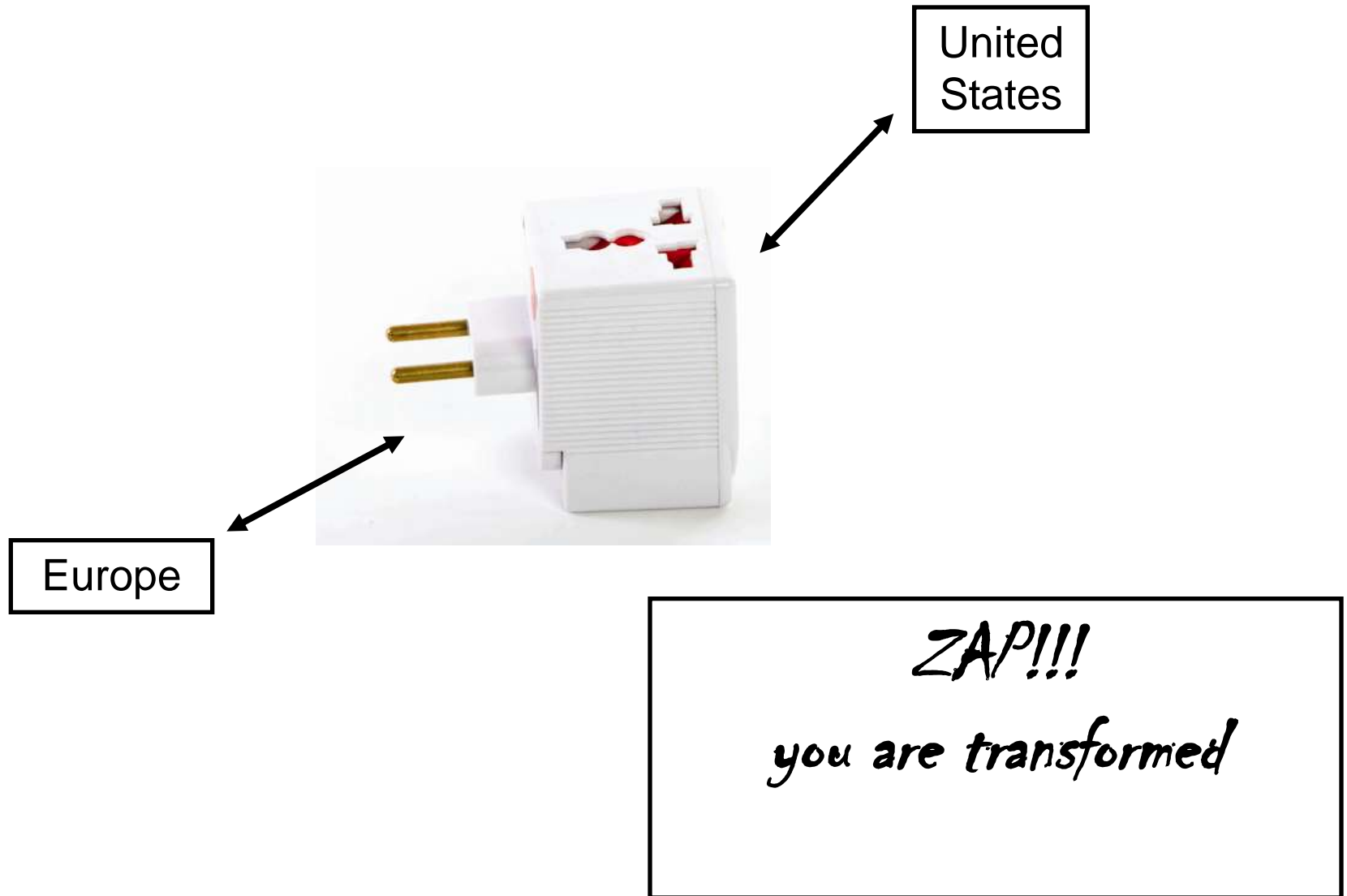
Welcome to CIA headquarters

The time is now:

08:00

It's **stringstream** time

Adapters



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 **strings**.

iostream and ostream

The `iostream` and `ostream`
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:

March 15, 2012

The code would be:

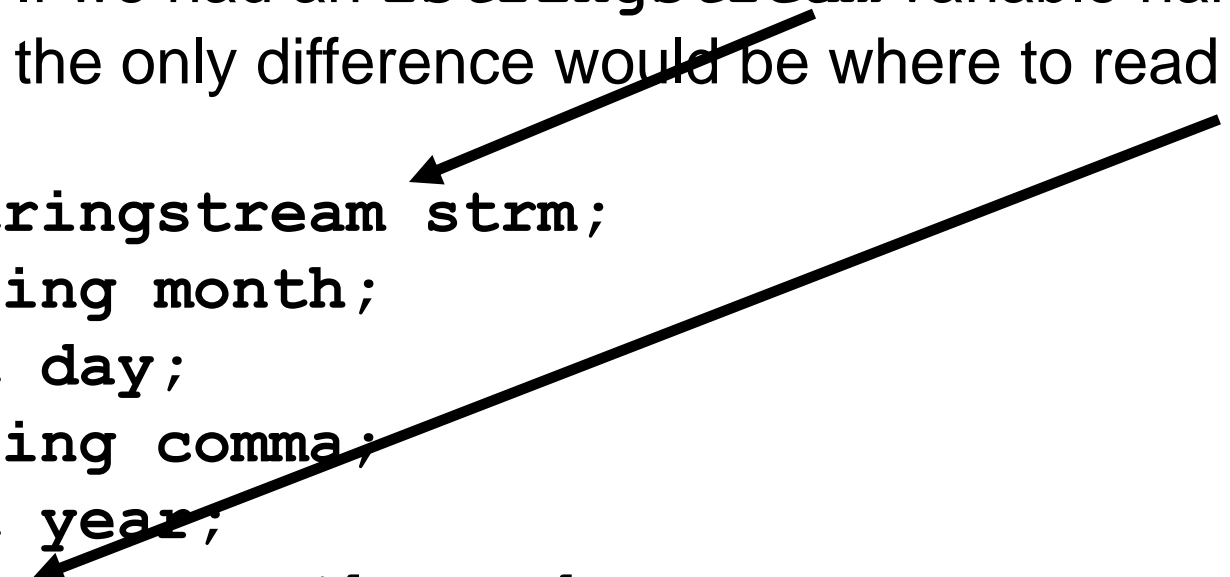
```
string month;  
int day;  
string comma;  
int year;  
cin >> month >> day >> comma >> year;
```

Normal.

The `istringstream` Type

What if that "March 15, 2012" were in a `string`?

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

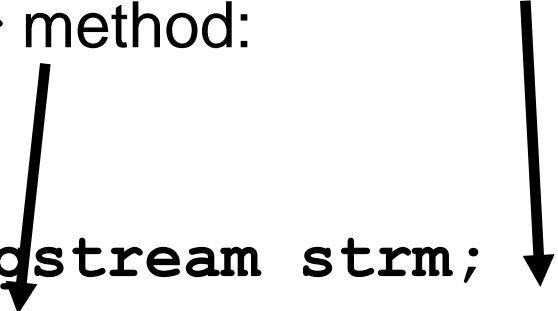


```
istringstream strm;  
string month;  
int day;  
string comma;  
int year;  
strm >> month >> day >> comma >> year;
```

No longer normal – but very nice!

The `istringstream` Type

To put "March 15, 2012" into a `string` you use the `str` method:



```
istringstream strm;  
strm.str("March 15, 2012");  
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?)

What's the opposite of extraction?

Insertion.

The “real” name of `>>` is *extraction (input stream operator)*
and `<<` is *insertion (output stream operator)*...

...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 = "March";  
int day = 15;  
int year = 2012;  
ostringstream strm;  
strm << month << " " << day << "," << year;  
    << " - "  
    << fixed << setprecision(5) << 10.0 / 3;  
string output = strm.str();
```

The `ostringstream` Type

A function for numbers to **strings**:

```
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:

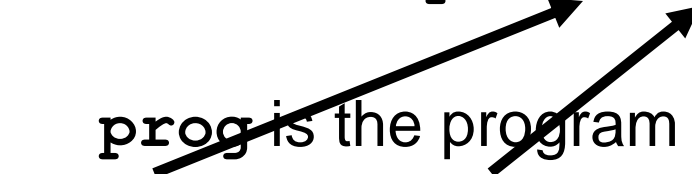
prog -v input.dat

 **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

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

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

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[])  
{  
    ...  
}
```

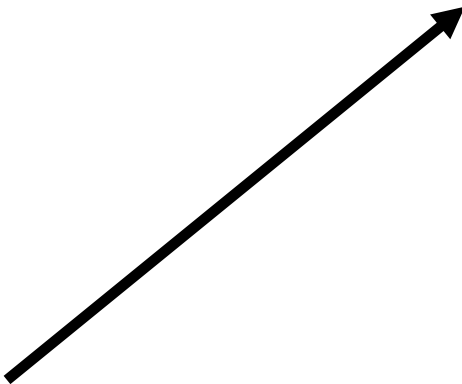


argc for argument count

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

Given that the user typed:

argv

prog	-v	input.dat
0	1	2

```
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"

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 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

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

Ahem...



but he's the emperor and we are not.

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!

I
made that



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

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

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(istream& in, ostream& out, int k)
```

```
{
```

```
    char ch;
```

```
    while (in.get(ch))
```

```
    {
```

```
        out.put(ch + k);
```

```
    }
```

```
}
```

The Fhdvdu Cipher



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; }
    }
```

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;
        }
    }
}
```

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;
    }
}
}
}
```


Programming: Encrypting A File

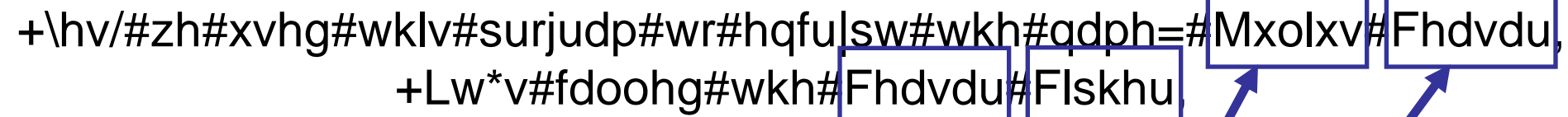
ch08/*fhdvdu*.cpp

```
// 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;
}
```

Programming: Encrypting A File

+\\hv/#zh#xvhg#wklv#surjudp#wr#hqfulsw#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)

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 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)



and don't forget ~~~~ we ~~~~

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?

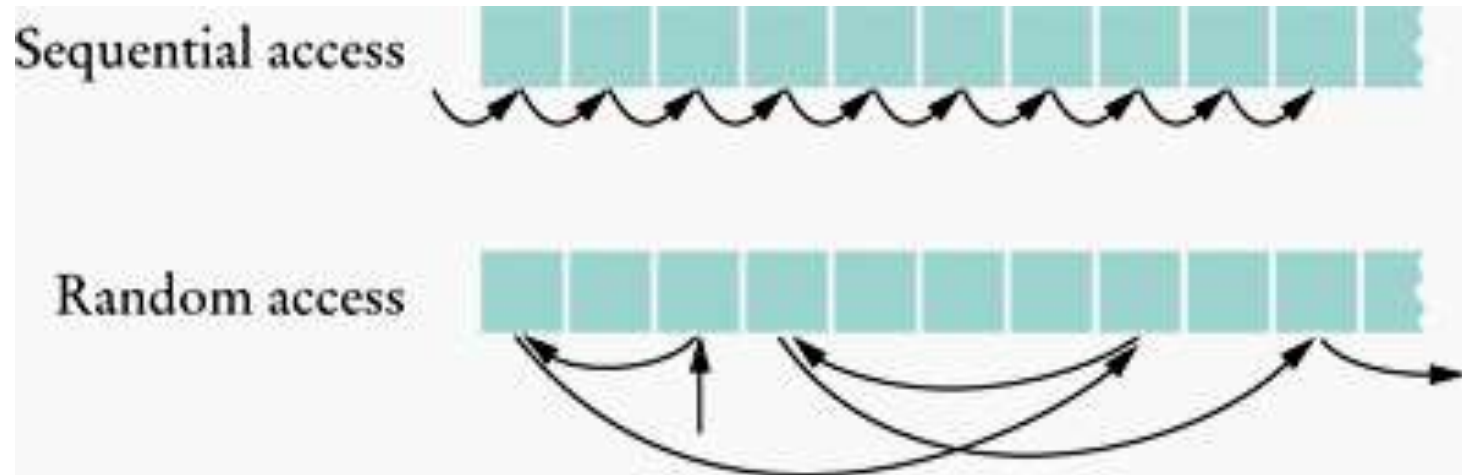
Random Access

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.

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?

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

Files have two special positions:

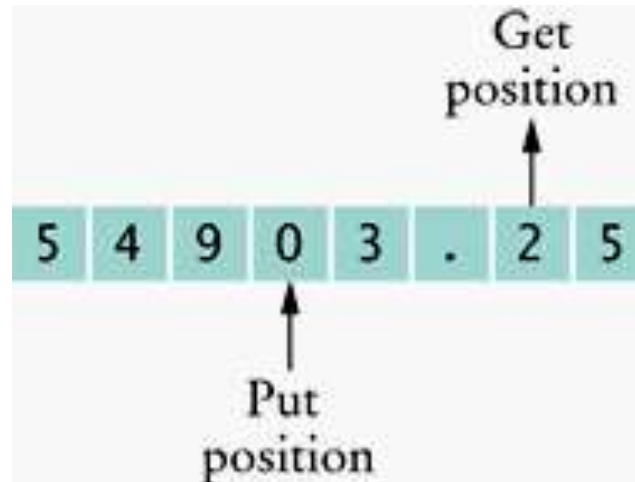
the *put* position – where the next write will go.

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

Random Access

The 0 will be overwritten next.

The 2 will be read next.



Random Access

To change the get positions, you use this method:

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



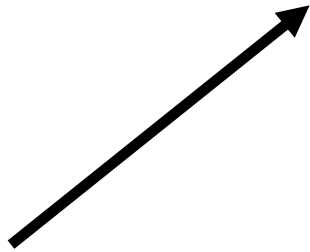
seekg means move the get position
“g” as in “get” – get it?

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

Random Access

`seekp` does the same for the put position

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



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 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 use the most simple of the
several versions of the BMP format:

the 24-bit true color format

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

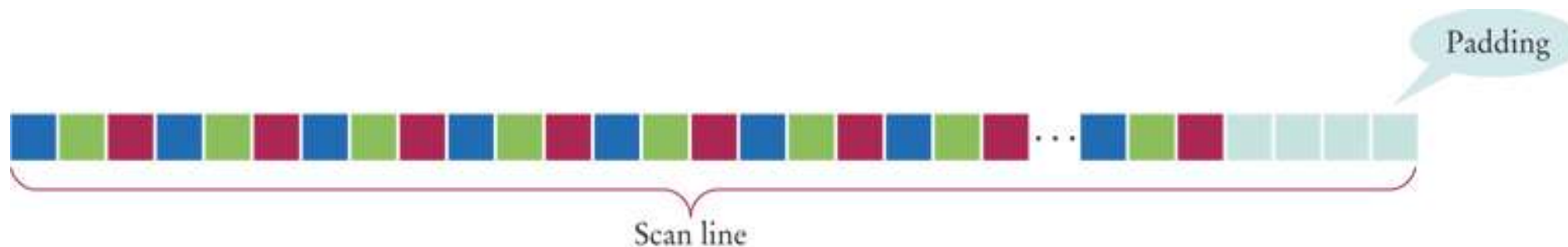


Processing Image Files: The BMP File Format

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.

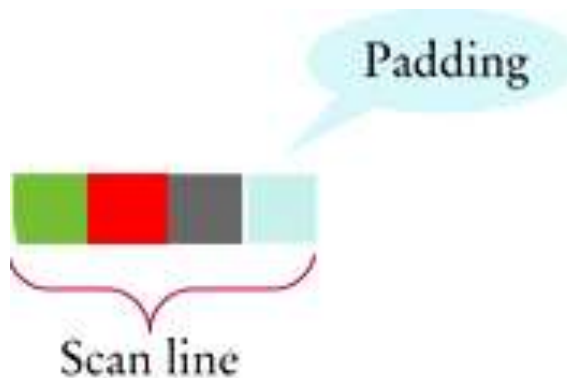


Processing Image Files: The BMP File Format

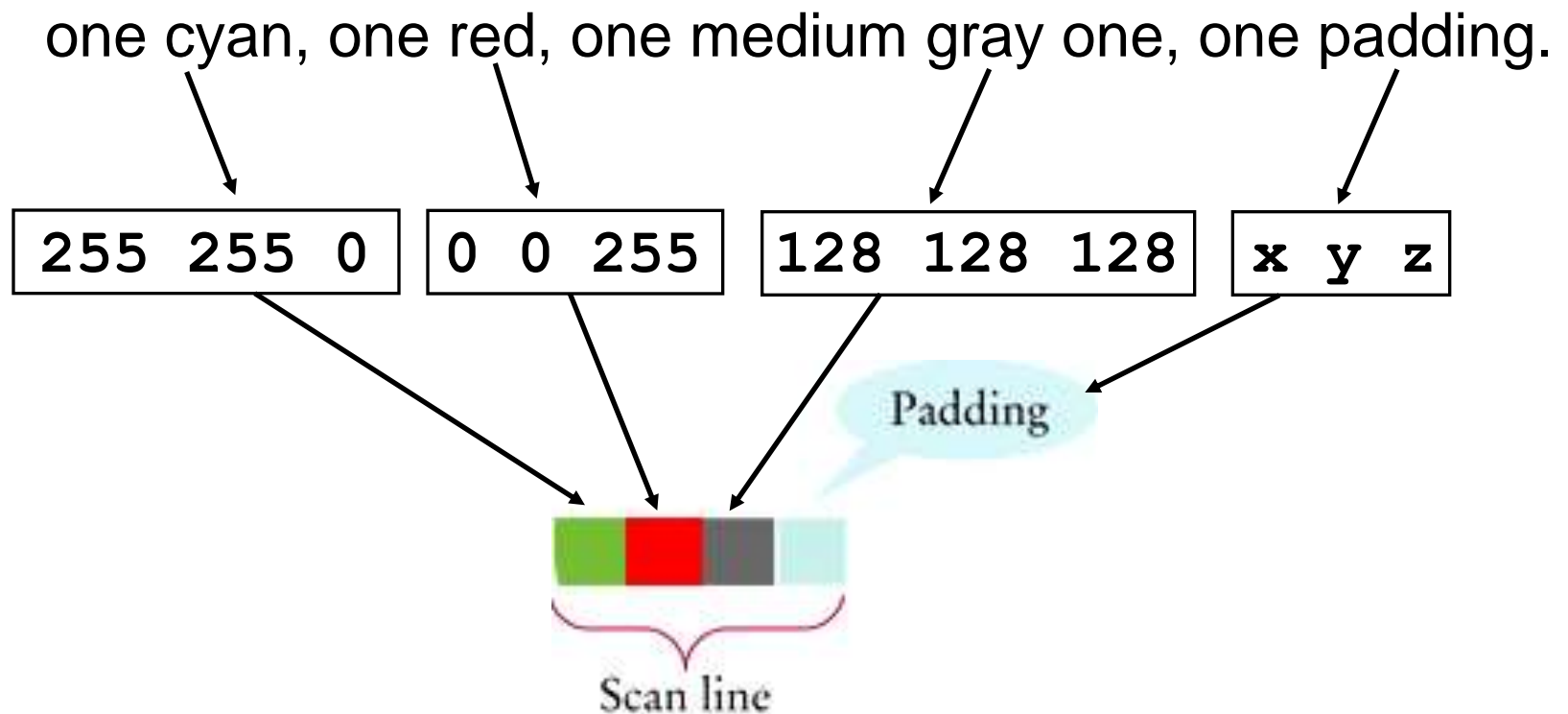
For example,
if a row consisted of merely three pixels,
one cyan, one red, one medium gray,
there would be three padding bytes.

The numbers would be:

255 255 0 0 0 255 128 128 128 **x y z**

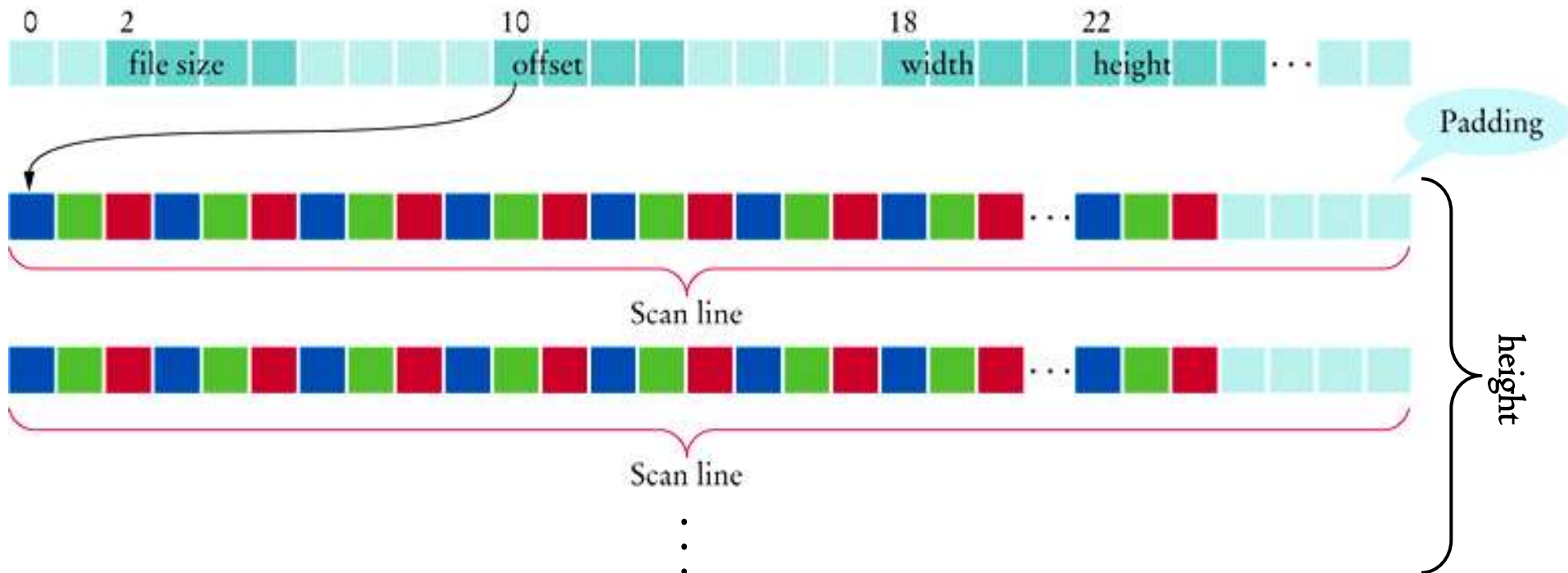


Processing Image Files: The BMP File Format



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

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

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: Streams, Part II