Topic 4

1. Reading and writing text files
2. Reading text input
3. Writing text output
4. Parsing and formatting strings
5. Command line arguments
6. Random access and binary files

Stream Adapters

In order to "extract" numeric information held in a **`string`**, we can *adapt* the **`string`** to have the same interface as stream types

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

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

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

`istringstream` Reads from a `string` into other Variables

Declare an **`istringstream`** variable named **`strm`** allows parsing data from a `string` such as

`"January 24, 1973"`

into numeric variables & other `strings` resembles `cin>>`:

```
istringstream strm;
//read string into the istringstream buffer
strm.str("January 24, 1973");

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

Converting a `string` to a single `int or double`

In C++11 and later, to convert a single string to its `int or double` value, use the functions `stoi` and `stod`:

```
string year = "1973";
int y = stoi(year);
string mass = "9.85";
double dmass = stod(mass);
```

Pre-C++11, write your own function using an `istringstream`:

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

The `ostringstream` Type for Building Strings

An **`ostringstream`** can copy text and numbers in a **`string`**. The numbers can be formatted as with files/cout, and the output operator << works the same.

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

// .str() "extracts" the stream into a string.
string output = strm.str();
```

Converting Numbers to `strings` without Formatting

If you don't need formatting, use the C++11 `to_string` function to convert numbers to strings:

```
string output = month + " " + to_string(day) + ", " +
   to_string(year);
```

Here is a function you can use with older versions:

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

Practice It: `istringstream` code

Complete the following function:

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

/**
   Decomposes an address into city, state, and zip.
   @param address an address such as "Ann Arbor, Michigan 48109"
   @return a vector<string> with city, state, and zip, such as
 { "Ann Arbor", "Michigan", "48109" }
*/
vector<string> decompose(string address)
{
   istringstream strm;
   strm.str(address);

   . . .
}
```

Topic 5

1. Reading and writing text files
2. Reading text input
3. Writing text output
4. Parsing and formatting strings
5. Command line arguments
6. Random access and binary files

Command Line

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”)
  - –To open this command shell in Windows, type “cmd” in the Search box, then click on cmd.exe
  - –The name of the program you type on the command line will depend on the IDE you used, but it might be your project name.

Command Line Arguments

In each of these methods, you can pass some information in via “command line arguments”

These arguments are passed to the **`main`** function!

Yes, the cmd shell program *calls the* ***`main`*** *function of your program.*

Command Line Arguments Example

The user might type into a command shell window for starting the program named **`prog`**:

```
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.

`main` is set up differently for Command Line Arguments

If your program intends to process command line arguments, **`main`** must be set up differently:

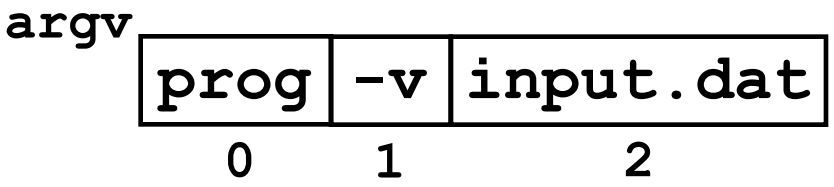
```
int main(int argc, char* argv[])
{
   ...
}
```

`argc` for argument count: **`argc`** =1 if the user typed nothing after your program name

`argv` for argument vector (not a "real" vector, but an array of `char` pointers, IE, C-strings)

Command Line Arguments: Example Values

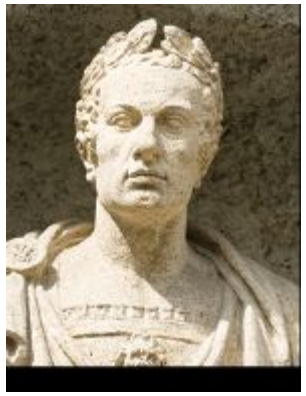
Given that the user typed:




```
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 Example Program: Encrypting A File



- Let's write a program that encrypts a file
  - scrambles it so that it is unreadable except to those who know the decryption method.
- Ignoring 2,000 years of progress in encryption, use a method familiar to Julius Caesar
  - replacing an A with a D, a B with an E, and so on.
  - each character c is replaced with c + 3

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

Programming: Encrypting A File

We will use the name *caesar* for our exe, with 3 command line arguments:

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

THUS OUR CODE WILL NOT PROMPT THE USER FOR FILE NAMES!! It will get them as the C-strings in `argv[]`.

For example,

```
caesar input.txt encrypt.txt
```

encrypts the file input.txt with the result in encrypt.txt.

```
caesar -d encrypt.txt output.txt
```

decrypts the file encrypt.txt and places the result into output.txt.

Command Line Program: Encrypting A File, Part 1

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

/**
   Encrypts a stream using the Ceasar 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);
   }
}
```

Command Line Program: Encrypting A File, Part 2

```
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; }
```

Command Line Program: Encrypting A File, Part 3

```
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;
      }
   }
```

Command Line Program: Encrypting A File, Part 4

```
      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;
         }
      }
   }
}
```

Command Line Program: Encrypting A File, Part 5

```
   // 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;
}
```

How To: Process Text Files, Example Program

Consider this task: Read two country data files, worldpop.txt and worldarea.txt (supplied with the book's companion code).

Both files contain the same countries in the same order.

Write a file world_pop_density.txt that contains country names and population densities (people per square km), with the country names aligned left and the numbers aligned right:

```
Afghanistan                    50.56
Akrotiri                      127.64
Albania                       125.91
Algeria                        14.18
American Samoa                288.92
. . .
```

How To: Process Text Files, Step by Step, Part 1

- Programs to process files can be complex. Here are the steps, adapted to the two-file task above:

1. Understand the processing task. The following pseudocode describes it:

```
While there are more lines to be read
  Read a line from each file.
  Extract the country name.
  population = number following the country name in the first line
  area = number following the country name in the second line
  If area != 0
    density = population / area
  Print country name and density.
```

2. Determine which files you need to read and write:
   - worldpop.txt and worldarea.txt, assumed to be in the same folder as the exe.
3. Choose a method for obtaining the file names. Options:
   1. Hard-coding the file names (such as "worldpop.txt")
   2. Asking the user
   3. Using command-line arguments for the file names

How To: Process Text Files, Step by Step, Part 2

4. Choose between line, word, and character-based input.
   - We'll use line input
5. With line-oriented input, extract the required data.
6. Place repeatedly occurring tasks into functions.

- a helper function: extracting the country name and the value that follows.
  - **`void read_line(string line, string& country, double& value)`**
- also a function **`string_to_double`** to convert population and area values to floating-point numbers

7. If required, use manipulators to format the output:

```
out << setw(40) << left << country << setw(15) << right <<
density << endl;
```

To view the complete program, see the textbook companion code `sec05/popdensity.cpp`. It is too long to repeat here.