CSE 1325

Week of 11/07/2022

Instructor : Donna French

Learning C++

- C++ can run differently depending on what machine you are using
- We will be using a standard setup that everyone will be required to use
- We will be using
  - C++ 11
  - Linux Ubuntu 64 bit

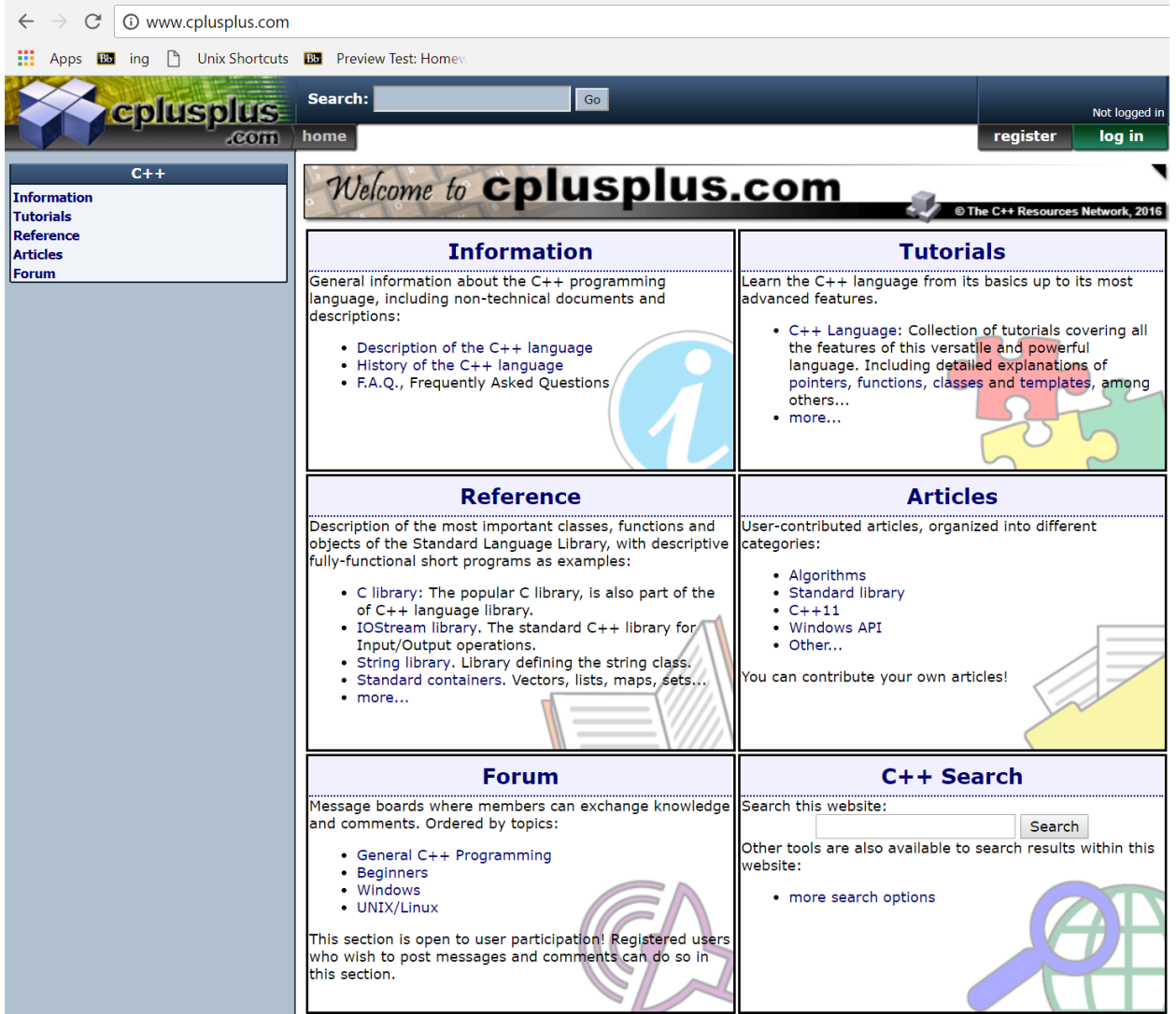
C++ Resources

www.cplusplus.com

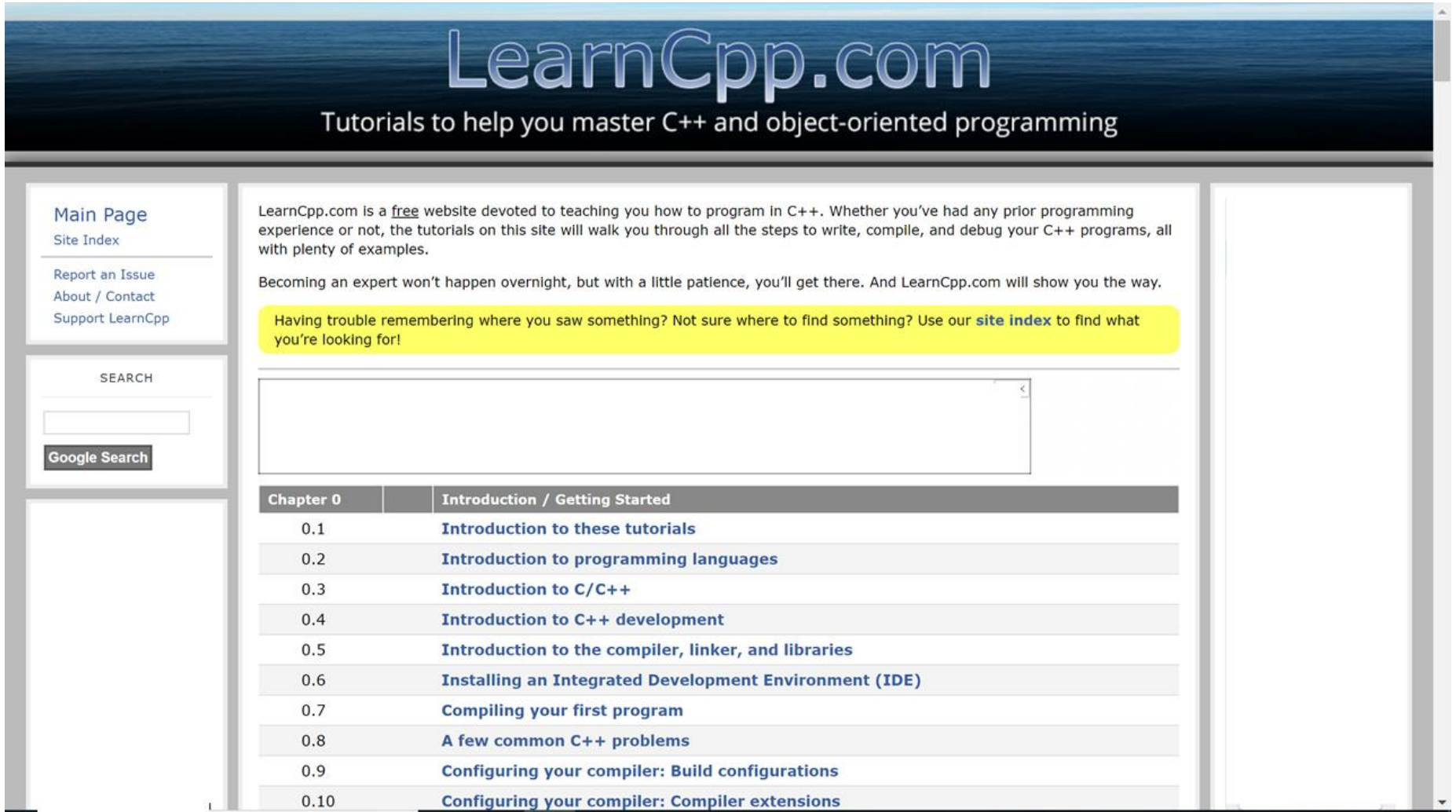
is a good resource

Other resources

- Stack Overflow
- O'Reilly books



Website for CSE 1325
www.learncpp.com



Variables in C++

Familiar variable types from C carry over to C++

```
char
short
int
float
double
void
long
unsigned
signed
```

These are built-in types

A new built-in type in C++

```
bool x
```

`x` is a Boolean which can have a value of true(1) or false(0)

New types defined in the standard library

```
string xxxx
```

`xxxx` is stream of characters

makefile

C++ uses a `makefile` just like C.

Change `.c` to `.cpp`

Change `gcc` to `g++`

```
#makefile for C++ program
SRC = HelloWorld.cpp
OBJ = $(SRC:.cpp=.o)
EXE = $(SRC:.cpp=.e)

CFLAGS = -g -std=c++11

all : $(EXE)

$(EXE): $(OBJ)
	g++ $(CFLAGS) $(OBJ) -o $(EXE)

$(OBJ) : $(SRC)
	g++ -c $(CFLAGS) $(SRC) -o $(OBJ)
```

Hello World

In C

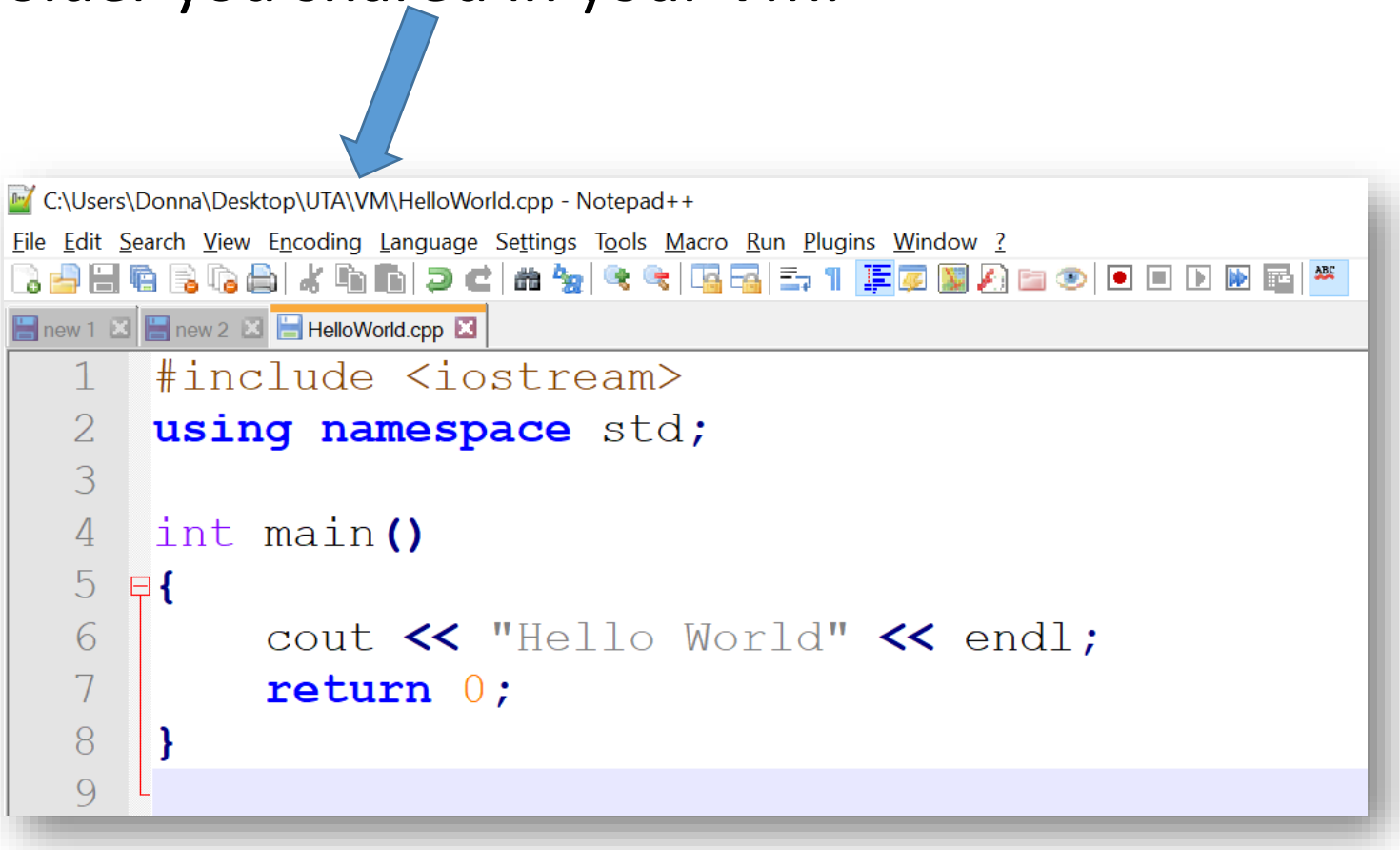
```
#include <stdio.h>
int main(void)
{
   printf("Hello World\n");
   return 0;
}
```

In C++

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

int main()
{
   cout << "Hello World" << endl;
   return 0;
}
```

Hello World

Use your favorite editor (I use Notepad++) to write HelloWorld.cpp.
Save to the folder you shared in your VM.



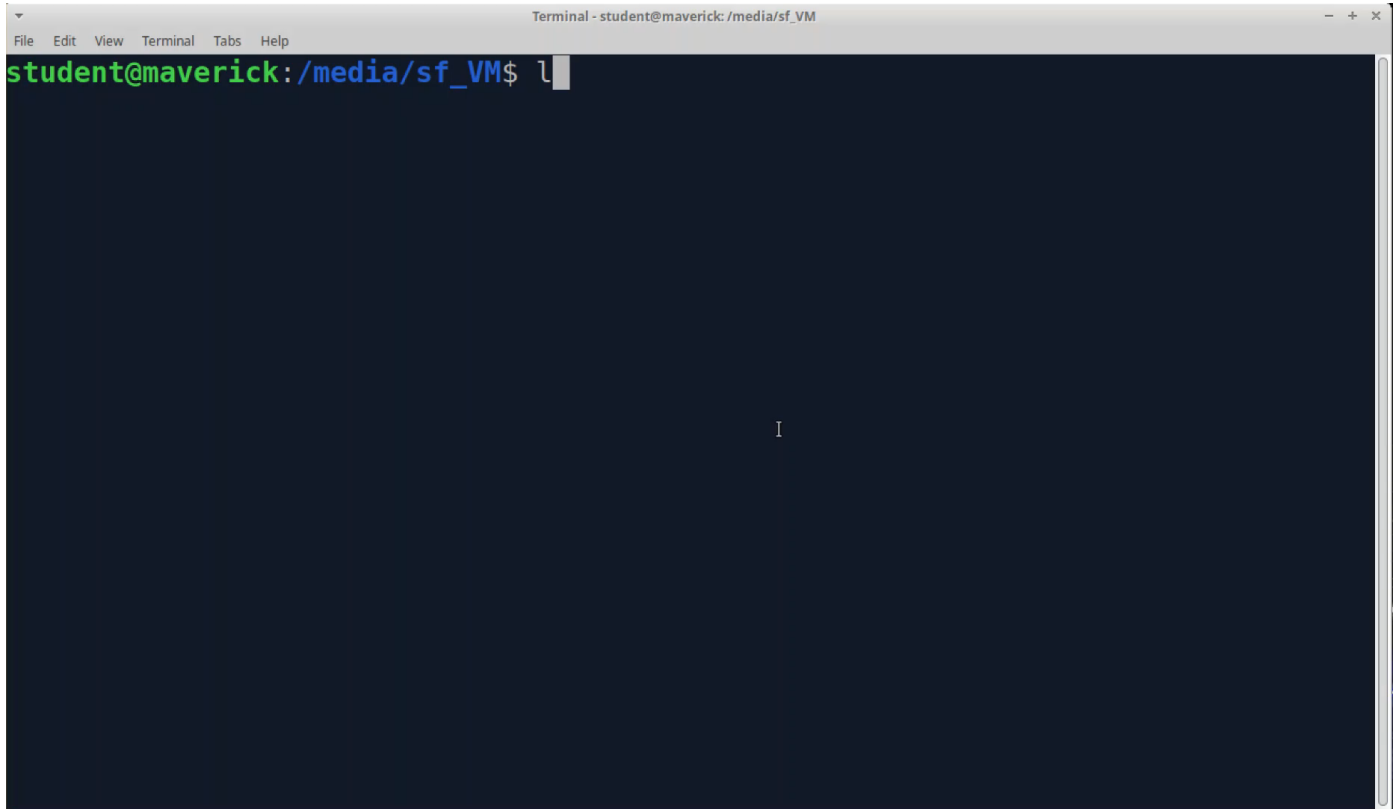
Hello World

You should be able to see it now in your VM when you open your shared folder with the terminal.

```
g++ HelloWorld.cpp
```

Should produce an a.out file.
Run your executable with

```
./a.out
```



Hello World

```
#include <iostream>
```

`iostream` is the header file which contains the functions for formatted input and output including `cout`, `cin`, `cerr` and `clog`.

C++ standard library packages don’t need a .h to reference them.

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World" << endl;
    return 0;
}
```

Hello World

`using namespace std`

The built in C++ library routines are kept in the standard namespace which includes `cout`, `cin`, `string`, `vector`, `map`, etc.

Because these tools are used so commonly, it's useful to add "`using namespace std`" at the top of your source code so that you won't have to type the **std**:: prefix constantly.

We use just

`cout`

instead of

`std::cout`

```
#include <iostream>

using namespace std;

int main()
{
   cout << "Hello World" << endl;
   return 0;
}
```

Hello World

What is a `namespace`?

`namespace` is a language mechanism for grouping declarations. Used to organize classes, functions, data and types.

Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World" << endl;
    return 0;
}
```

I could create a function with the same name and define its own namespace and use the :: scope resolution operator to refer to my version.

Hello World

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World" << endl;
    return 0;
}
```

`cout` and `<<` and `endl`

`cout` is an abbreviation of **c**haracter **out**put stream.

`<<` is the output operator

`endl` puts `'\n'` into the stream and flushes it

So the line

```
cout << "Hello World" << endl;
```

puts the string “Hello World” into the character output stream and flushes it to the screen

Hello World Plus

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

int main()
{
    string first_name;
    cout << "Hello World" << endl;
    cout << "What is your name?" << endl;
    cin >> first_name;
    cout << "Hello " << first_name << endl;
    return 0;
}
```

`string` is a variable type that can hold character data

`cin` is an abbreviation of **c**haracter **in**put stream.

`>>` is the input operator

Hello World Plus

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

int main()
{
    string first_name;
    cout << "Hello World" << endl;
    cout << "What is your name?" << endl;
    cin >> first_name;
    cout << "Hello " << first_name << endl;
    return 0;
}
```

This line

```
cin >> first_name;
```

puts whatever you type at the terminal (up to the first whitespace) into the string variable `first_name`

Note that the <ENTER> key (newline) is not stored in `first_name`

Standard Stream Objects

`cin`

- `istream` object
- "connected to" the standard input device
- uses stream extraction operator >>

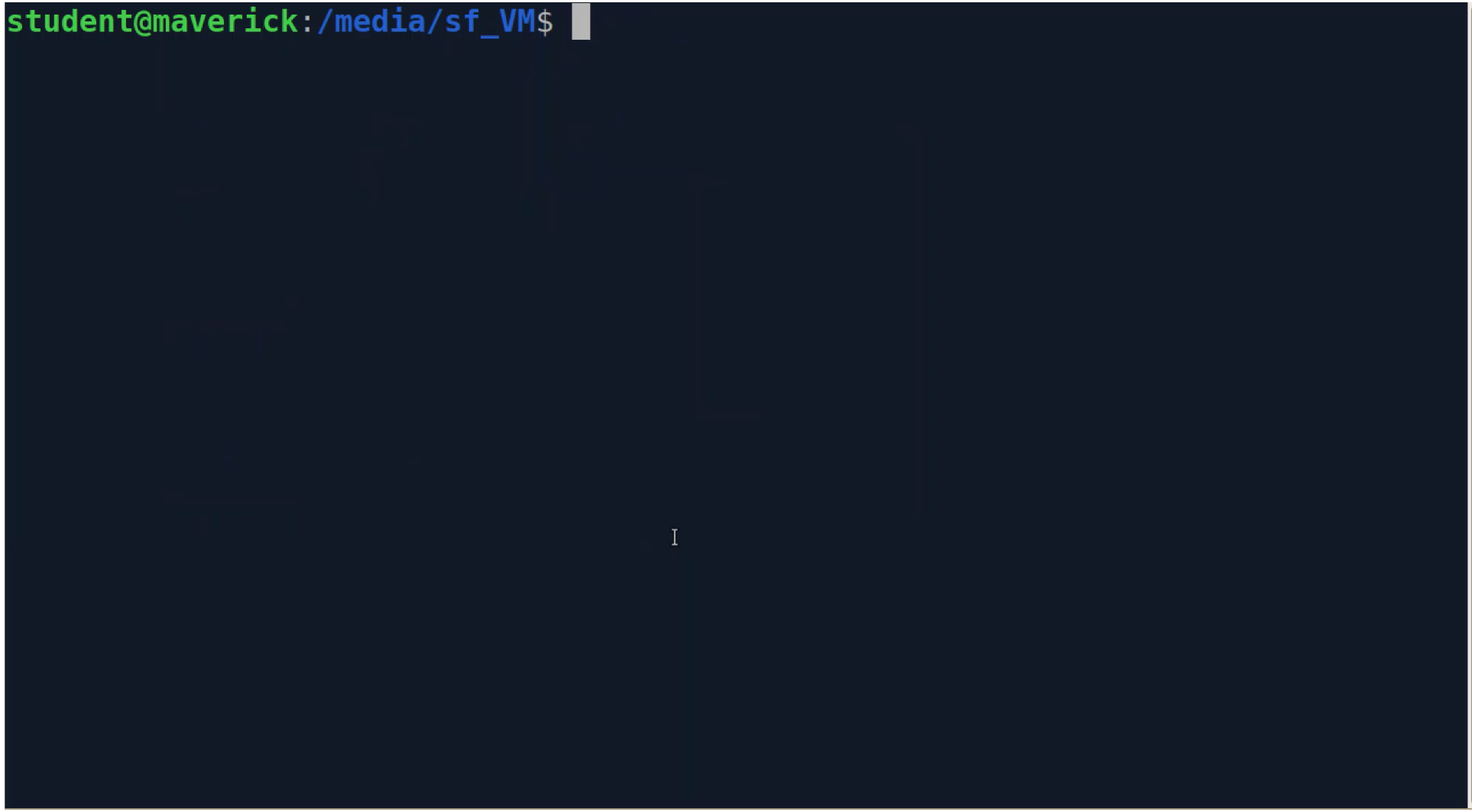
```
int grade;
cin >> grade;
```

`cout`

- `ostream` object
- "connected to" the standard output device
- uses stream insertion operator <<

```
cout << grade;
```

Hello World Plus



cin

```
cin >> CreamPuff;
```

cout

```
cout << “Happy Birthday”;
```

stream insertion
vs
stream extraction

<<

stream insertion operator

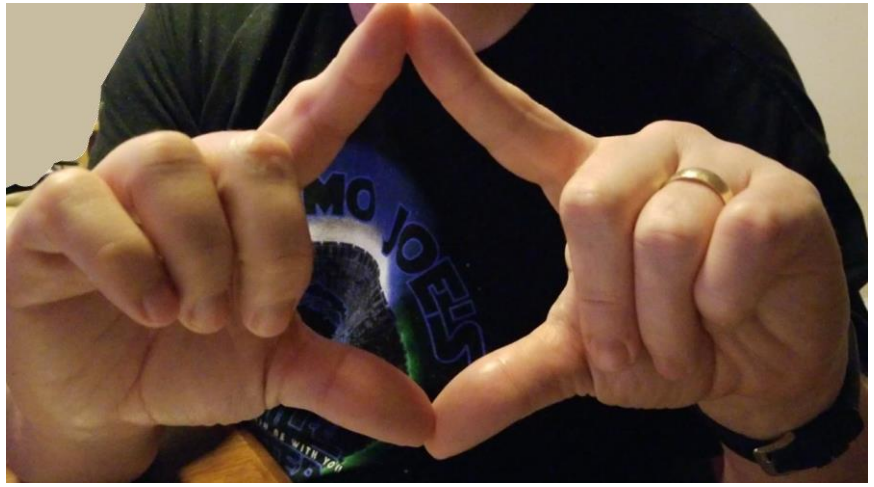
>>

stream extraction operator

Remember the rule in English of "i before e except after c"?

i before e

insertion	extraction
<<	>>



std::string

Just like the Java and C, a string is a collection of sequential characters.

C++ has a `string` type. Just like the Java, `string` is actually an object; therefore, knows things and can do things. This will make more sense once we start talking about classes and member functions.

To use `string` include the `string` header file.

```
#include <string>
```

std::string

As we did with `cin` and `cout`, we can either put

```
using namespace std
```

in our .cpp file and not need to preface `string` with `std::` or we can not use the `std` namespace and need to use `std::string`.

```
std::string MyString;
string MyString;
```

std::string

We've already seen the example where `cin` stops reading at whitespace (just like `scanf()`).

```
string first_name, last_name, full_name;


cout << "Hello!\n" << endl;
cout << "What is your name? (Enter your first name and last name) " << endl;
cin >> first_name >> last_name;
cout << "Hello " << first_name << ' ' << last_name << endl;
```

std::string

What if we need to read a line of input including the whitespace into a single variable?

For example, what if I wanted to take whatever name was entered and only store it in one variable?

```
string full_name;


cout << "Hello!\n" << endl;
cout << "What is your name? " << endl;
cin >> full_name;
cout << "Hello " << full_name << endl;
```

If I type

```
Fred Flintstone
```

at the prompt, what will print?

std::string

`getline()` is the C++ version of `fgets()` from C. It takes two parameters just like `fgets()`.

The first parameter is the stream to read from – when reading from the screen use `cin`.

The second parameter is `string` variable where you want to store the input.

```
string full_name;

cout << "Hello!\n" << endl;
cout << "What is your name? " << endl;
getline(cin, full_name);
cout << "Hello " << full_name << endl;
```

```
Hello!

What is your name?
Fred Flintstone
Hello Fred Flintstone
```

std::string

Mixing `cin` with `getline()` can cause issues

`cin` leaves the newline (`\n`) in the standard input buffer.

```
10          cin >> dog_name;
(gdb)
What is your dog's name? Dino
11          cout << "Hi " << dog_name << endl;
(gdb) p *stdin
$1 = {_flags = -72539512, _IO_read_ptr = 0x555555769284 "\n",
```

std::string

Which `getline()` then reads and uses; therefore, not prompting for more input.

We can use

```
cin.ignore(50, '\n');
```

This function discards the specified number of characters or fewer characters if the delimiter is encountered in the input stream.

Puts a null at the end of the buffer and throws out the newline

New keywords in C++

`const`

Used to inform the compiler that the value of a particular variable should not be modified.

If a value does not (or should not) change in the body of a function to which it's passed, the parameter should be declared `const`.

```
const int counter = 1;
```

`counter` is an integer constant

Pass by Reference in C++

C++ has a specific syntax for passing by reference.

To indicate that a function parameter is passed by reference, follow the parameter's type in the function prototype by an ampersand (&); use the same convention when listing the parameter's type in the function header.

```
int& number
```

`number` is a reference to an `int`

```
int main(void)
{
  int MyMainNum = 0;

  cout << "Before PassByRefCPlusPlus   call\tMyMainNum = " << MyMainNum << endl;
  PassByRefCPlusPlus(MyMainNum);
  cout << "After  PassByRefCPlusPlus   call\tMyMainNum = " << MyMainNum << endl;

  return 0;
}

int PassByRefCPlusPlus(int& MyNum)
{
  MyNum += 100;
  cout << "Inside PassByRefCPlusPlus\t\tMyNum     = " << MyNum << endl;
}
```

simrefDemo.cpp

Pass By Reference
Pros vs Cons

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

Pass-by-reference can weaken security; the called function can corrupt the caller’s data.

`const` References

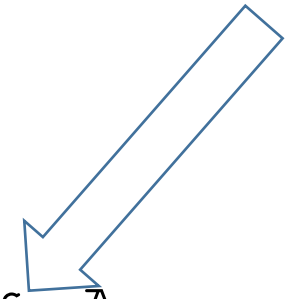
Function `setName` uses pass-by-value.

```
void setName(std::string AccountName)
{
   string name = AccountName;
}
```

When this function is called, it receives a copy of its `string` argument. `string` objects can be large, so this copy operation degrades an application's performance.

`const` References

For this reason, `string` objects (and objects in general) should be passed to functions by reference.



```
void setName(std::string& AccountName)
{
    name = AccountName;
}
```

`const` References

But, this means that the function can change/corrupt the data.

To specify that a reference parameter should not be allowed to modify the corresponding argument, place the `const` qualifier before the type name in the parameter's declaration.

```
void setName(const std::string& AccountName)
{
    name = AccountName;
}
```

We get the performance of passing the string by reference, but `setName` treats the argument as a constant, so it cannot modify the value in the caller—just like with pass-by-value. Code that calls `setName` would still pass the string argument exactly as before.

Streams

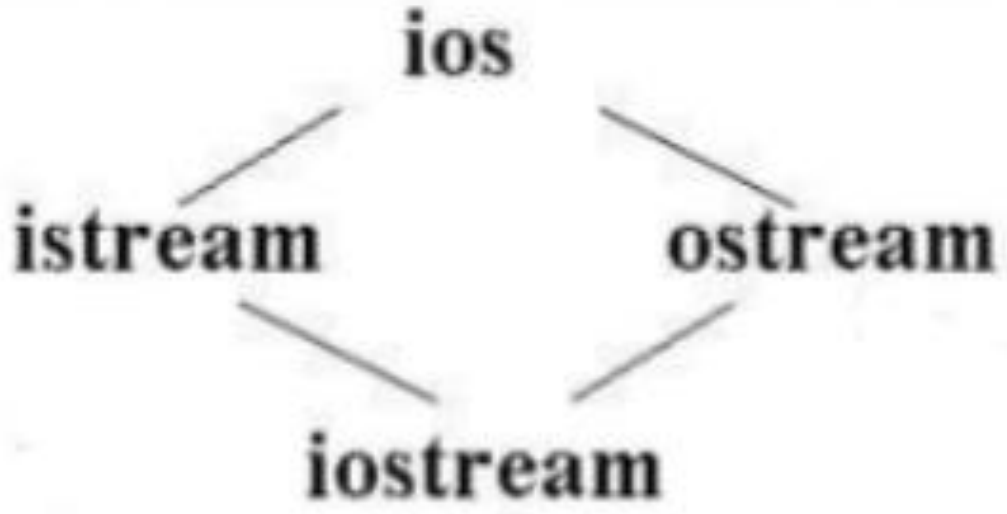
- C++ I/O occurs in streams of bytes
- A stream is a sequence of bytes
  - Input – bytes flow from a device (e.g., keyboard, drive) to memory
  - Output – bytes flow from memory to a device (e.g., screen, printer)
- C++ provides
  - low-level I/O capabilities
    - unformatted
    - high speed and high volume
  - high-level I/O capabilities
    - formatted
    - people friendly
    - bytes are grouped into meaningful units (integers, floats, characters, strings, etc)
      - type-oriented capabilities

Stream Libraries

- iostream
  - contains objects that perform basic I/O on standard streams
    - cin
    - cout
- iomanip
  - contains objects that perform formatted I/O with stream manipulators
- fstream
  - contains objects that perform user-controlled file processing operations
- strstream
  - contains objects that perform memory formatting

Streams
iostream library

- istream class
  - supports stream-input operations
- ostream class
  - supports stream-output operations
- iostream class
  - supports both stream-input and stream-output operations



Streams
Operator Overloading

<<

left shift operator is overloaded to be the stream-insertion operator

`cout`

object of ostream class
tied to standard output
assumes type of data

```
cout << "Hello!”;
```

>>

right shift operator is overloaded to be the stream-extraction operator

`cin`

object of istream class
tied to standard input
assumes type of data

```
string first_name,last_name;
cin >> first_name >> last_name;
```

Streams
Operator Overloading

C++ determines data types automatically – does not require the programmer to supply the type information

```
printf("%s", MyString);
printf("%d", MyInt);
printf("%f", MyDouble);
```

```
cout << MyString;
cout << MyInt;
cout << MyDouble;
```

Sometimes, this gets in the way…

PrintAddress1Demo.cpp

Streams
Operator Overloading

The << operator has been overloaded to print data of type char* as a null terminated string. That won't result in the address of a pointer.

```
char MyChar = 'A';
char *MyPtr = &MyChar;

printf("Value of MyChar    %c\n", MyChar);
printf("Address of MyChar  %p\n", MyPtr);

cout << "Value of MyChar " << MyChar << endl;
cout << "Value of MyPtr " << MyPtr << endl;

cout << "Address of MyChar " << (void *)MyPtr << endl;
```

```
Value of MyChar A
Value of MyPtr A_????
```

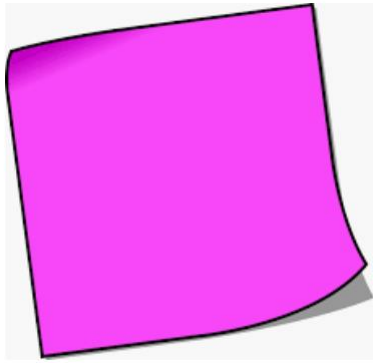
Stream Manipulators

C++ uses stream manipulators to perform formatting tasks

- setting field widths
- setting precision
- setting and unsetting format flags
- setting the fill character in fields
- flushing streams
- inserting a newline in the output stream and flushing the stream
- inserting a null character in the output stream
- skipping whitespace in the input stream

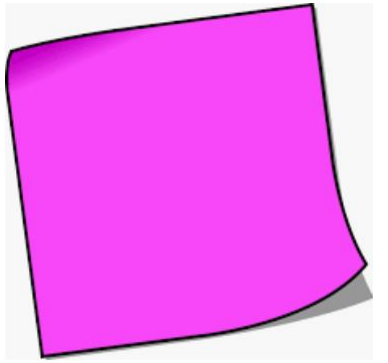
Sticky vs. Non-Sticky Stream Manipulators

A sticky stream manipulator permanently changes stream behavior - permanently until the next change, that is.



A non-sticky stream manipulator only effects the stream for the next value.

Stream Manipulators
Integers



`dec`, `oct`, `hex`, `showbase` **and** `setbase`

Integers are normally interpreted as decimal (base 10) values

This interpretation can be altered by inserting a manipulator into the stream.

These only affect integers – using them with other types will have no effect.

```
int MyIntA = 10, MyIntB = 20, MyIntC = 30;


cout << showbase;
cout << "none    " <<        MyIntA << "\t" << MyIntB << "\t" << MyIntC << endl;
cout << "decimal " << dec << MyIntA << "\t" << MyIntB << "\t" << MyIntC << endl;
cout << "hex     " << hex << MyIntA << "\t" << MyIntB << "\t" << MyIntC << endl;
cout << "octal   " << oct << MyIntA << "\t" << MyIntB << "\t" << MyIntC << endl;
cout << "\n\n\n";
```

```
none    10      20      30
decimal 10      20      30
hex     0xa     0x14    0x1e
octal   012     024     036
```

iomanipDemo.cpp

```
cout << oct << MyIntA << "\t" << dec << MyIntB << "\t" << hex << MyIntC << endl;
cout << noshowbase;
cout << oct << MyIntA << "\t" << dec << MyIntB << "\t" << hex << MyIntC << endl;

cout << "\n\n\n";
cout << setbase(8)  << MyIntA << " "
     << setbase(10) << MyIntB << " "
     << setbase(16) << MyIntC << endl;
```

`setbase()` can be called using a variable

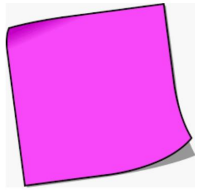
`setbase(basevalue)`

`setbase()` might be better to use since it can be passed a value.

Rather than hardcoding `hex`, `oct`, `dec`, you can just use one `setbase()`

`iomanipDemo.cpp`

Stream Manipulator `boolalpha`



```
#include <iostream>

using namespace std;

int main()
{
  cout << "false && false\t" << (false && false) << endl;

  cout << boolalpha
       << "true  || false\t" << (true || false) << endl;

  cout << noboolalpha
       << "true  ^  true\t" << (true ^ true) << endl;

  return 0;
}
```

```
false  &&  false      0
true   ||  false      true
true   ^   true       0
```

false – keyword that evaluates to zero

true – keyword that evaluates to non-zero

Stream
Error State Flags

Each stream object contains a set of **state bits** that represent a stream's state

Stream extraction

- sets the stream's failbit to true if the wrong type of data is input.
- sets the stream's badbit to true if the operation fails in an unrecoverable manner - for example, if a disk fails when a program is reading a file from that disk.

Stream – Error State Flags

`eof`

- member function of iostream
- used to determine whether end-of-file has been encountered on the stream
- checks the value of the stream's `eofbit` data member
  - set to TRUE for an input stream after end-of-file is encountered after an attempt to extract data beyond the end of the stream
  - set to FALSE if EOF has not been reached

```
cout << "Error State Flags before a bad input operation " << endl
     << "\ncin.eof()      " << cin.eof()
     << "\ncin.fail()     " << cin.fail()
     << "\ncin.good()     " << cin.good()
     << "\ncin.bad()      " << cin.bad();
```

Stream – Error State Flags

`fail`

- member function of iostream
- used to determine whether a stream operation has failed
- checks the value of the stream's `failbit` data member
  - set to TRUE on a stream when a format error occurs and, as a result, no characters are input
    - when asking for a number and a string is entered
- when `fail()` returns TRUE, the characters are not lost

```
cout << "Error State Flags before a bad input operation " << endl
     << "\ncin.eof()      " << cin.eof()
     << "\ncin.fail()     " << cin.fail()
     << "\ncin.good()     " << cin.good()
     << "\ncin.bad()      " << cin.bad();
```

Stream – Error State Flags

`good`

- member function of iostream
- used to determine whether a stream operation has failed
- checks the value of the stream's `goodbit` data member
  - set to TRUE for a stream if none of the bits `eofbit`, `failbit` or `badbit` is set to true for the stream

```
cout << "Error State Flags before a bad input operation " << endl
     << "\ncin.eof()      " << cin.eof()
     << "\ncin.fail()     " << cin.fail()
     << "\ncin.good()     " << cin.good()
     << "\ncin.bad()      " << cin.bad();
```

Stream – Error State Flags

`bad`

- member function of iostream
- used to determine whether a stream operation has failed
- checks the value of the stream's `badbit` data member
  - set to TRUE for a stream when an error occurs that results in the loss of data
    - reading from a file when the disk on which the file is stored fails
- indicates a serious failure that is nonrecoverable

```
cout << "Error State Flags before a bad input operation " << endl
     << "\ncin.eof()      " << cin.eof()
     << "\ncin.fail()     " << cin.fail()
     << "\ncin.good()     " << cin.good()
     << "\ncin.bad()      " << cin.bad();
```

Stream – Error State Flags

After an error occurs, you can no longer use the stream until you reset its error state

`clear`

- member function of iostream
- used to *restore* a stream's state to "good" so that I/O may proceed on that stream
- clears `cin` and sets `goodbit` for the stream

```
cin.clear();
```

```
cout << "Error State Flags before a bad input operation " << endl
     << "\ncin.eof()      " << cin.eof()
     << "\ncin.fail()     " << cin.fail()
     << "\ncin.good()     " << cin.good()
     << "\ncin.bad()      " << cin.bad();
```

```
cin.eof()      0
cin.fail()     0
cin.good()     1
cin.bad()      0
```

```
cout << "\n\nEnter a character to cause cin to fail on reading an int ";

cin >> IntVar;

cout << "\n\nError State Flags after a bad input operation " << endl
     << "\ncin.eof()      " << cin.eof()
     << "\ncin.fail()     " << cin.fail()
     << "\ncin.good()     " << cin.good()
     << "\ncin.bad()      " << cin.bad();
```

```
cin.eof()      0
cin.fail()     1
cin.good()     0
cin.bad()      0
```

esfDemo.cpp

```
cin.clear();

cout << "\n\nError State Flags after the clear operation " << endl
     << "\ncin.eof()      " << cin.eof()
     << "\ncin.fail()     " << cin.fail()
     << "\ncin.good()     " << cin.good()
     << "\ncin.bad()      " << cin.bad() << endl;
```

```
cin.eof()      0
cin.fail()     0
cin.good()     1
cin.bad()      0
```

esfDemo.cpp

Stream – Error State Flags

`cin` uses the error state flags to terminate a while loop

Input failure

```
int grade;
while (cin >> grade)
{
}
```

```
cin.eof()    0
cin.fail()   1
cin.good()   0
cin.bad()    0
```

EOF encountered

```
string MySentence;
while (cin >> MySentence)
{
}
```

```
cin.eof()    1
cin.fail()   1
cin.good()   0
cin.bad()    0
```

Streams
Input

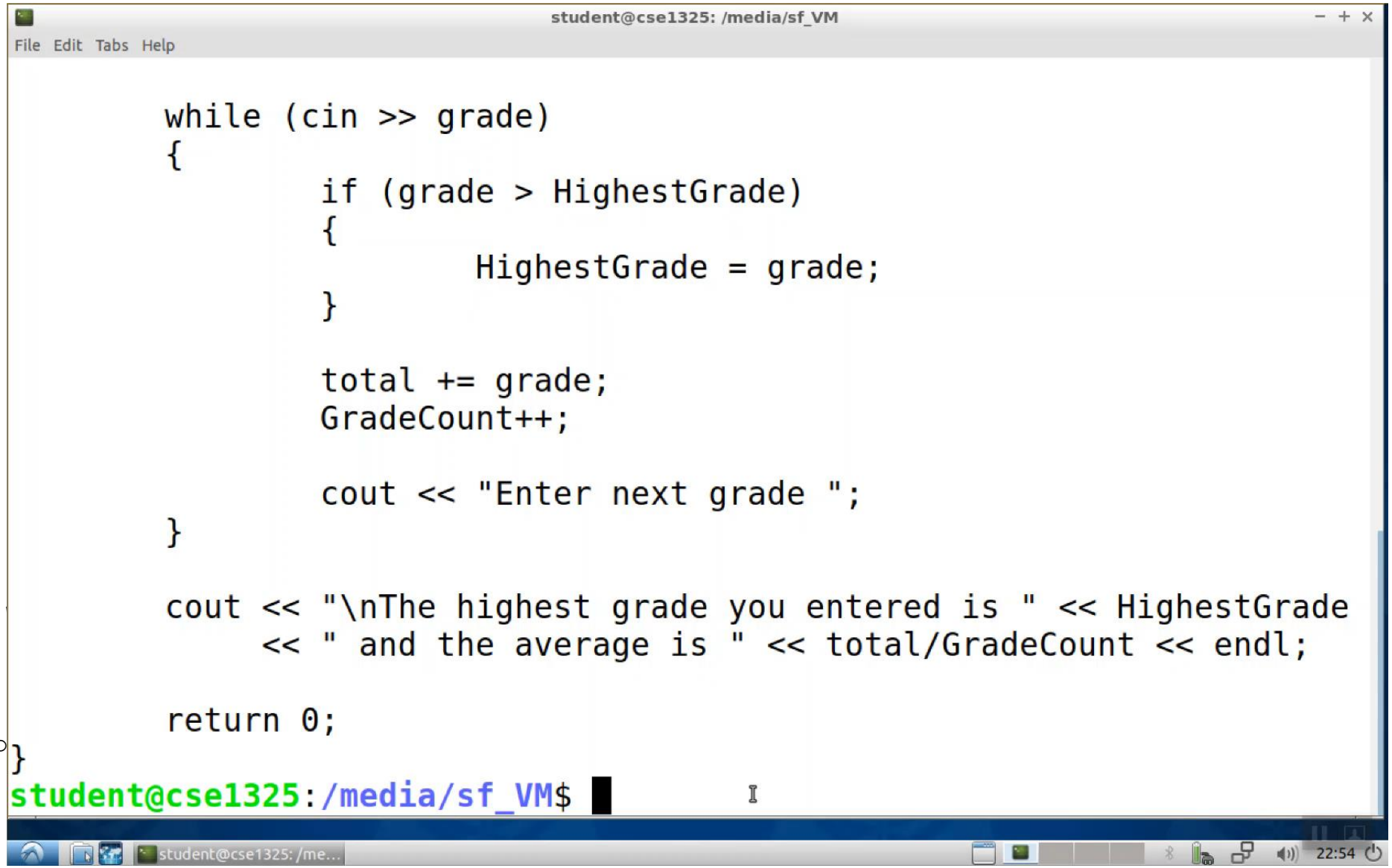
Using `cin` as the condition of a `while` loop

```
while (cin >> grade)
```

Why does this work?

The input to `cin` is converted into a pointer of type `void *`. The value of that pointer is 0 if an error occurred while attempting to read a value or when it reads the EOF indicator. Returning a 0 gives `while` a FALSE causing the condition to fail and the loop to stop.

```
int grade, GradeCount = 0, HighestGrade = -1;
double total = 0;

cout << "Enter each grade ";

while (cin >> grade)
{
     if (grade > HighestGrade)
     {
          HighestGrade = grade;
     }

     total += grade;
     GradeCount++;

     cout << "Enter next grade
}

cout << "\nThe highest grade yo
      << HighestGrade
       << " and the average is "
      << total/GradeCount;
```



whilecout1Demo.cpp

`std::cin`

When we use operator >> to get user input and put it into a variable, this is called an “extraction”.

The >> operator is called the extraction operator when used in this context.

When the user enters input in response to an extraction operation, that data is placed in a buffer.

std::cin

When the extraction operator is used, the following procedure happens:

- If there is data already in the input buffer, that data is used for extraction.
- If the input buffer contains no data, the user is asked to input data for extraction (this is the case most of the time). When the user hits <ENTER>, a '\n' character will be placed in the input buffer.
- operator >> extracts as much data from the input buffer as it can into the variable (ignoring any leading whitespace characters, such as spaces, tabs, or '\n').

Any data that cannot be extracted is left in the input buffer for the next extraction.

Stream Summary

- C++ I/O occurs in streams which are sequences of bytes
- I/O operations are sensitive to the data type
- `<iostream>` header – all stream I/O operations
- `<iomanip>` header – parameterized stream manipulators
- `istream`
  - `cin` object
- `ostream`
  - `cout` object
- The state of a stream can be tested

File Processing

C++ stream I/O includes capabilities for writing to and reading from files.

Class `ifstream`

Supports file input (reading from a file)

Class `ofstream`

Supports file output (writing to a file)

Class `fstream`

Supports file input/output (writing to/reading from a file)

Header file `<fstream>` must be included in addition to `<iostream>`

File Processing – Opening a File

Open a file for output by creating an `ofstream` object (calling a constructor)

Two arguments

- filename
- file open mode

```
ofstream MyOutputFileStream{"outfile.txt", ios::out};
```

File Processing – File Open Modes

Ios file mode	Meaning
app	Opens the file in append mode
ate	Seeks to the end of the file before reading/writing
binary	Opens the file in binary mode (instead of text mode)
in	Opens the file in read mode (default for ifstream)
out	Opens the file in write mode (default for ofstream)
trunc	Erases the file if it already exists

File Processing – Opening a File

After opening a file, check if the open was successful

`is_open()`

member function of `ofstream`

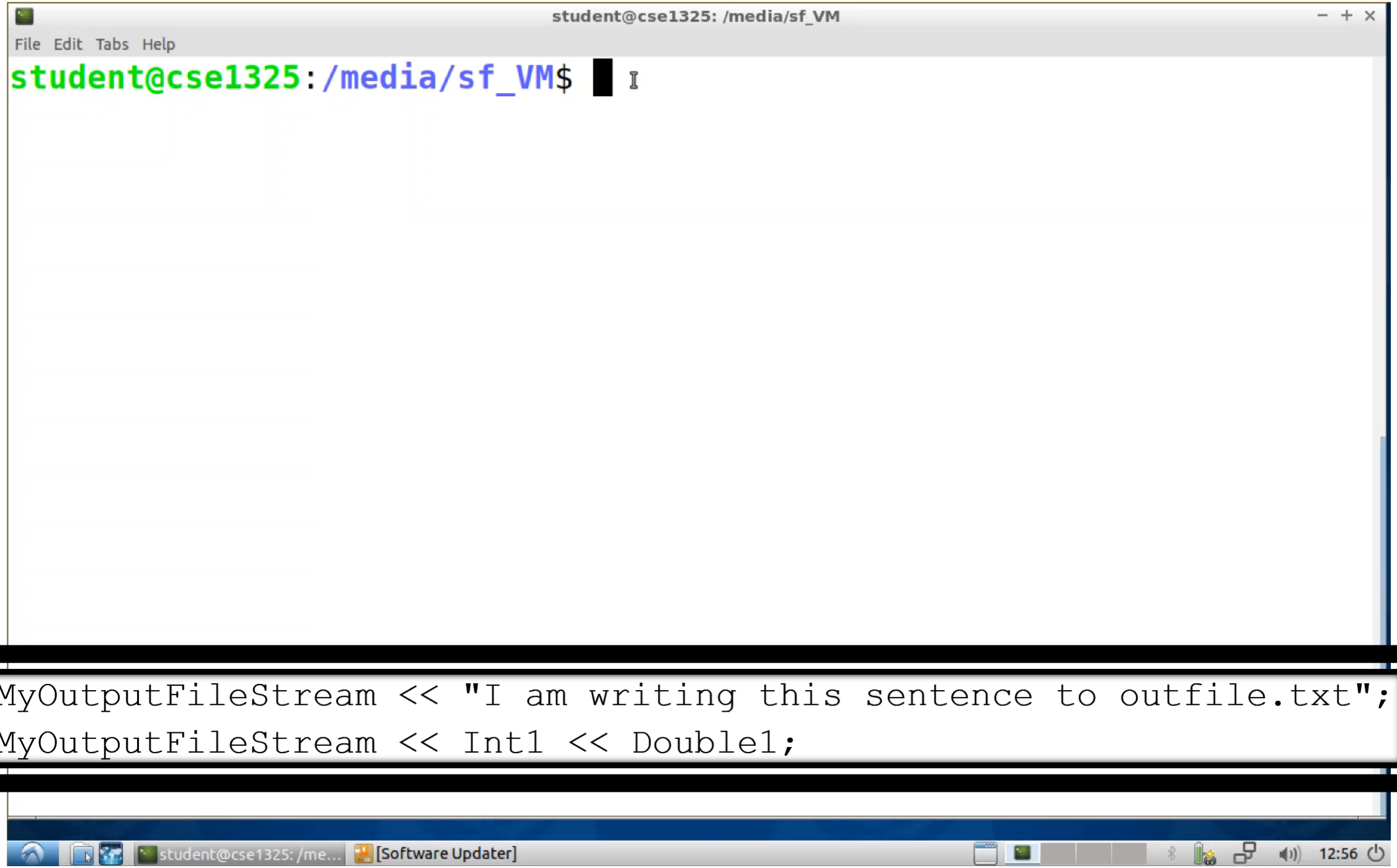
returns TRUE if file is open and associated with given stream and FALSE if it is not

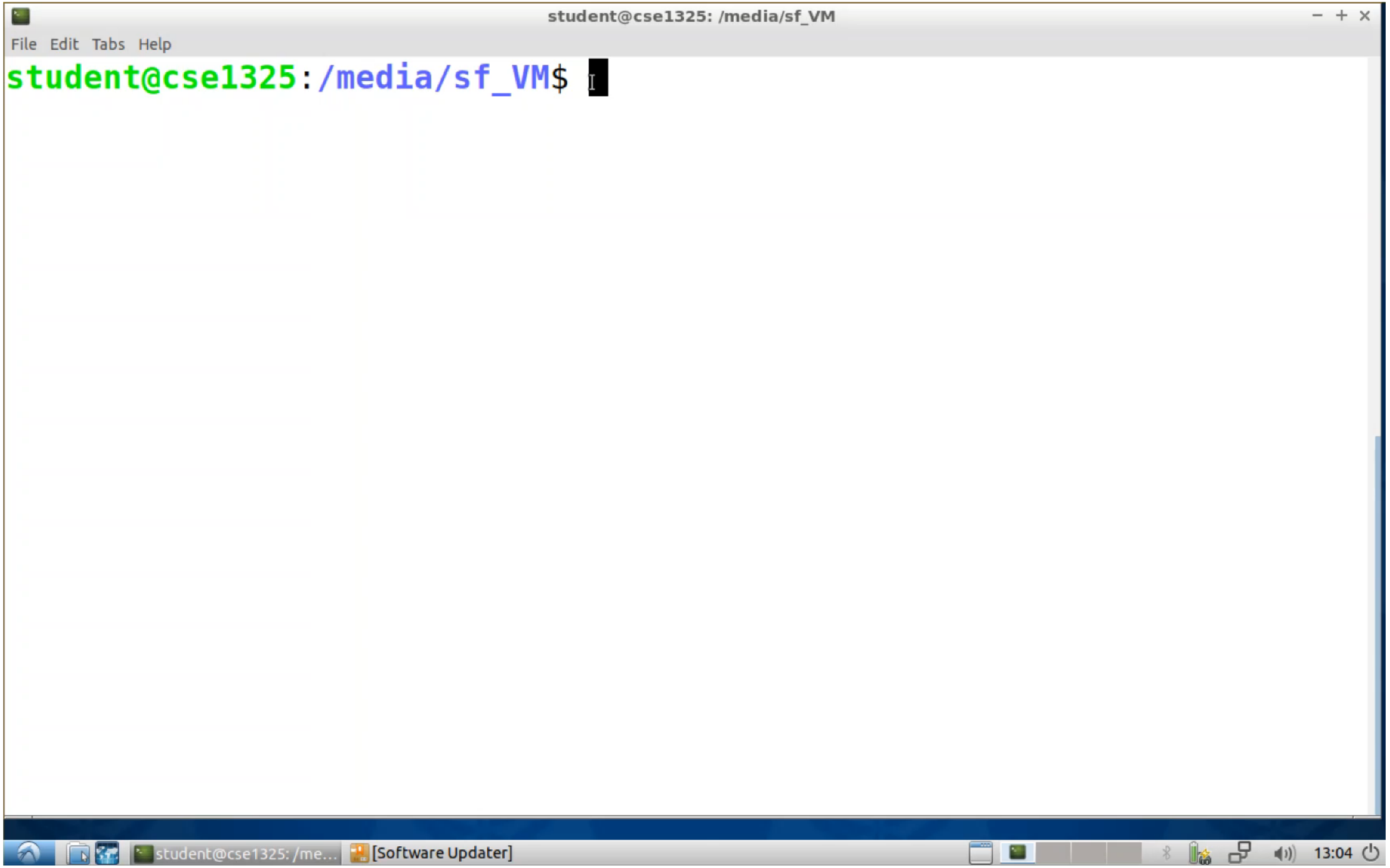
```
if (MyOutputFileStream.is_open())
{
   cout << "The file opened" << endl;
}
else
{
   cout << "The file did not open" << endl;
}
```

File Processing – Writing to a File

```
ofstream MyOutputFileStream{"outfile.txt", ios::out};
int Int1 = 10;
double Double1 = 12.34;

if (MyOutputFileStream.is_open())
{
   MyOutputFileStream << "I am writing this sentence to outfile.txt";
   MyOutputFileStream << Int1 << Double1;
}
else
{
   cout << "The file did not open" << endl;
}

MyOutputFileStream.close();
```

ofstream1Demo.cpp

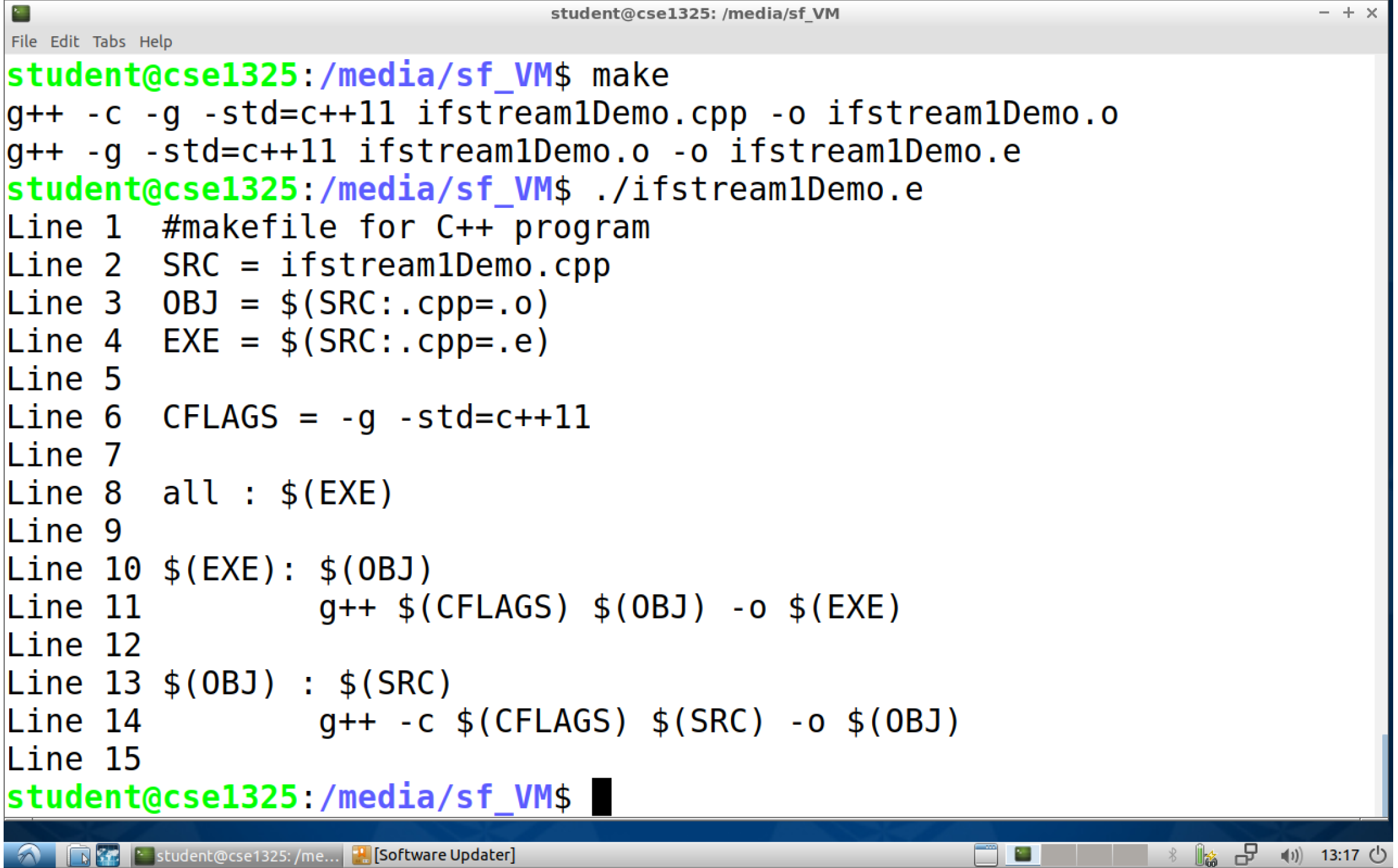

student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$
MyOutputFileStream << "I am writing this sentence to outfile.txt";
MyOutputFileStream << Int1 << Double1;
student@cse1325: /me...
[Software Updater]
12:56


student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$
student@cse1325: /me...
[Software Updater]
13:04

File Processing – Reading from a File

```
ifstream MyInputFileStream{"makefile"};
string MyLine;
int LineCounter = 0;

if (MyInputFileStream.is_open())
{
   while (getline(MyInputFileStream, MyLine))
   {
      cout << "Line " << ++LineCounter << "\t" << MyLine << endl;
   }
}
else
{
   cout << "The file did not open" << endl;
}

MyInputFileStream.close();
```

ifstream1Demo.cpp


student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 ifstream1Demo.cpp -o ifstream1Demo.o
g++ -g -std=c++11 ifstream1Demo.o -o ifstream1Demo.e
student@cse1325:/media/sf_VM$ ./ifstream1Demo.e
Line 1 #makefile for C++ program
Line 2 SRC = ifstream1Demo.cpp
Line 3 OBJ = $(SRC:.cpp=.o)
Line 4 EXE = $(SRC:.cpp=.e)
Line 5
Line 6 CFLAGS = -g -std=c++11
Line 7
Line 8 all : $(EXE)
Line 9
Line 10 $(EXE): $(OBJ)
Line 11   g++ $(CFLAGS) $(OBJ) -o $(EXE)
Line 12
Line 13 $(OBJ) : $(SRC)
Line 14   g++ -c $(CFLAGS) $(SRC) -o $(OBJ)
Line 15
student@cse1325:/media/sf_VM$
student@cse1325: /me...
[Software Updater]
13:17

```
char MyChar;
int DigitCounter = 0;

ifstream MyPhoneNumberF

if(MyPhoneNumberFile.is
{
   cout << "eofbit  is
   cout << "goodbit is

   while (MyPhoneNumber
   {
      if (isdigit(MyCha
      {
         cout.put(MyCha
         if (!(++DigitC
            cout << endl
       }
   }

   cout << "\n\neofbit
   cout << "goodbit is
}
else
   cout << "Unable to open file";
```

```
student@cse1325:/media/sf_VM$ more PhoneNumbers.txt
817a415b0687
21c47722d387
907d3f429811
student@cse1325:/media/sf_VM$ ./ifstream2Demo.e
eofbit  is 0
goodbit is 1

8174150687
2147722387
9073429811

eofbit  is 1
goodbit is 0
student@cse1325:/media/sf_VM$
```

ifstream2Demo.cpp

File Processing – Closing a File

When `main()` terminates, the `ofstream` destructor is implicitly called and the file is closed.

Good coding style is to close your own files as soon as you are done using them. In a production environment, files are shared by many processes and should be opened only when needed and closed as soon as possible to prevent conflicts with other processes.

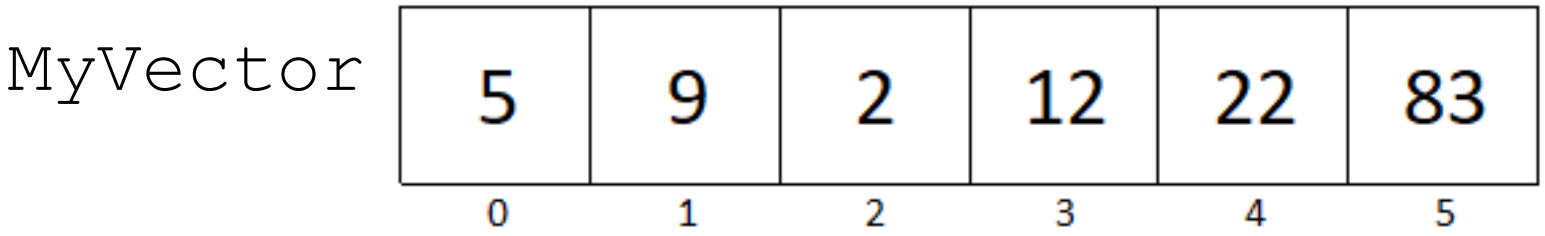
```
MyOutputFileStream.close();
```

You want to avoid holding files open unnecessarily in a shared environment because other programs – maybe hundreds of other programs may need that same file.

vector

Simple and useful way to store data

A vector is a sequence of elements that you can access by an index



`MyVector[0]` is 5

`MyVector[4]` is 22

vector

- Need to add an include to use vectors

```
#include <vector>
```

- Declaring a vector

```
vector<type> vectorname;
vector<type> vectorname(number of elements);
```

- Initializing and declaring a vector

```
vector<type> vectorname{comma delimited list of elements};
```

vector

How would you declare

a `bool vector` named `Cat` initialized to false, true, true, true, false

```
vector<bool>Cat{0,1,1,true,0};
```

vector

```
13                vector<char>Frog;
(gdb)
14                vector<float>Toad(7);
(gdb)
15                vector<bool>Cat{0,1,1,true,0};
(gdb)

(gdb) p Frog
$2 = std::vector of length 0, capacity 0
(gdb) p Toad
$3 = std::vector of length 7, capacity 7 = {0, 0, 0, 0, 0, 0, 0}
(gdb) p Cat
$4 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}
```

vector

A vector knows its size

```
vectorname.size()

vector<int> MyVector{2,4,6,8};
cout << "MyVector has " << MyVector.size() << " elements\n\n";

for (int i = 0; i < MyVector.size(); ++i)
     cout << MyVector[i] << endl;
```

vector

It is common to process *all* the elements of a vector.

The C++11 range-based for statement allows you to do this *without using a counter*,

This statement avoids the possibility of “stepping outside” the vector and eliminating the need for bounds checking.

When processing all elements of a vector, if you do not need to access to a vector element’s subscript, use the range based for statement.

vector

for loop

```
vector<int> MyVector = {2,4,6,8};
int i;
for (i = 0; i < MyVector.size(); i++)
    cout << setw(5) << MyVector[i];
```

```
  2    4    6    8
```

range-for-loop

```
vector<int> MyVector = {2,4,6,8};

for (int x : MyVector)
    cout << setw(5) << x;
```

```
 2    4    6    8
```

for each iteration, assign the next element of `MyVector` to `int` variable `x`, then execute the following statement

foreachDemo.cpp

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> ABunch {1,2,3,4,5,6,7,8,9,10};

    for (int i = 0; i < ABunch.size(); i++)
    {
        cout << "ABunch[" << ABunch[i] <<"]" << endl;
    }

    // read as "for each int banana in ABunch
    for (int banana : ABunch)
    {
        cout << "ABunch[" << banana << "]" << endl;
    }
}
```

```
vector <int> MyList{1,2,3,4,5,6};

cout << hex;

for (int it : MyList)
    cout << setw(10) << it;

cout << "\nMultiple every element of vector by 3" << endl;

for (int &it : MyList)
    it *= 3;

cout.fill('.');

for (int it : MyList)
    cout << setw(10) << it;
```

```
         1         2         3         4         5         6
Multiple every element of vector by 3
.........3.........6.........9.........c.........f.........12
```

```
#include <iostream>
#include <vector>

int main()
{
	std::vector<std::string>CatNames{"Shade", "Appa", "Sylvester", "Josie"};

	for (auto it : CatNames)
	{
		std::cout << it << "\t";
	}

    return 0;
}
```

```
Shade  Appa   Sylvester    Josie
```

vector

`push_back()`

- member function of `vector` (like `size()`)
- adds a new element to the end of the vector

```
vector<int> MyVector = {2,4,6,8};

MyVector.push_back(10);
MyVector.push_back(12);
```

push_back()

```
      2     4     6     8
```

```
The size of MyVector is 4 and the capacity of MyVector is 4
```

```
MyVector after push_back(10)
```

```
MyVector.push_back(10);
```

```
for (int x : MyVector)
    cout << x << "\t";
```

```
      2     4     6     8     10
```

```
The size of MyVector is 5 and the capacity of MyVector is 8
```

```
MyVector after push_back(12)
```

```
MyVector.push_back(12);
```

```
      2     4     6     8     10    12
```

```
The size of MyVector is 6 and the capacity of MyVector is 8
```

front() and back()

Vector member function front() returns the value stored in the first element of the vector

Vector member function back() returns the value stored in the last element of the vector.

```
vector <float> Bank{1.2345,2.3456,3.4567,4.5678,5.6789};

cout << fixed << Bank.front() << endl
     << scientific << Bank.back() << endl;
```

```
1.234500
5.678900e+00
```


```
MyVector.pop_back();

cout << "\n\nMyVector after pop_back()" << endl;

for (int x : MyVector)
    cout << x << "\t";

cout << "\nMyVector.size() "    << MyVector.size()
     << " MyVector.capacity() " << MyVector.capacity() << endl;
```

```
MyVector after pop_back()
2    4    6    8    10
MyVector.size() 5 MyVector.capacity() 8
```

pop_back()

Vector member function pop_back() removes the last element of the vector.

```
vector <float> Bank{1.2345,2.3456,3.4567,4.5678,5.6789};

Bank.pop_back();
```

```
for (float it : Bank)
    cout << it << setw(7);
```

```
1.2345 2.3456 3.4567 4.5678 5.6789
size = 5 and capacity = 5
```

```
cout << "size = " << Bank.size()
     << " and capacity = "
     << Bank.capacity() << endl;
```

```
1.2345 2.3456 3.4567 4.5678
size = 4 and capacity = 5
```

erase()

```
MyVector.erase(MyVector.begin()+1);

cout << "\n\nMyVector after erase()" << endl;

for (int x : MyVector)
    cout << x << "\t";

cout << "\nMyVector.size() "    << MyVector.size()
     << " MyVector.capacity() " << MyVector.capacity() << endl;
```

```
MyVector after erase()
2     6     8     10
MyVector.size() 4 MyVector.capacity() 8
```

begin() vs front()

```
MyVector.erase(MyVector.begin()+1);
MyVector.erase(MyVector.front()+1);

vector3Demo.cpp: In function ‘int main()’:
vector3Demo.cpp:53:35: error: no matching function for call to
‘std::vector<int>::erase(__gnu_cxx::__alloc_traits<std::allocator<int
> >::value_type)’
  MyVector.erase(MyVector.front()+1);


 erase(const_iterator __position)
       ^~~~~
/usr/include/c++/7/bits/stl_vector.h:1179:7: note:   no known
conversion for argument 1 from
‘__gnu_cxx::__alloc_traits<std::allocator<int> >::value_type {aka
int}’ to ‘std::vector<int>::const_iterator _Alloc>::const_iterator =
```

`begin()` vs `front()`

Definition of `begin()`

```
const_iterator begin() const noexcept;
```

Returns an iterator pointing to the first element in the vector.

Definition of `front()`

```
const_reference front() const;
```

Returns a reference pointing to the first element in the vector.

Definition of `erase()`

```
iterator erase (const_iterator position);
```

`position`

Iterator pointing to a single element to be removed from the vector.

at()

`at(n)` returns a reference to the element at position n in the vector

```
vector <string> States{"Indiana", "Oklahoma", "Texas"};
cout << States.at(2);
```

```
Texas
```

Remember that we start counting at 0

at()

```
vector <string> States{"Indiana", "Oklahoma", "Texas"};

cout << "The list of states" << endl;

for (i = 0; i < 3; i++)
{
    cout << i+1 << ". " << States[i+1] << "\t";
}
```

```
The list of states
Segmentation fault (core dumped)
```

at()

```
vector <string> States{"Indiana", "Oklahoma", "Texas"};

cout << "The list of states" << endl;

for (i = 0; i < 3; i++)
{
    cout << i+1 << ". " << States.at(i+1) << "\t";
}
```

```
The list of states
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check: __n (which is 3) >= this->size()
(which is 3)
Aborted (core dumped)
```

Operations on a vector

`size()`

`capacity()`

`front()`

`back()`

`at(`*n*`)`

`pop_back()`

`erase(`*n*`)`

`begin(n)`

`end(n)`

vector

So did all this discussion on vectors make you think of something from C?

A stack in C++ can be implemented using a vector and

- `push_back()` pushes an element on the stack
- `back()` returns the value of the top element on the stack
- `pop_back()` pops an element off the stack