CSE 1325

Week of 09/21/2020

Instructor : Donna French

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

- C++ imposes no structure on files
- The concept of a record does not exist in C++
- The program/programmer must enforce a definition on the stream of data

File Processing

Stream of data in a file

```
Richard  Tiffany  Gere         Kobe      Bean      Bryant       Elton
Hercules John         Ben      Geza      Affleck      Matt      Paige    Damon
```

If we impose a structure on this stream of

characters 1 - 9 = First name

characters 10 - 19 = Middle name

characters 20 – 31 = Last name

```
          1         2         3
12345678901234567890123456789 01
Richard  Tiffany   Gere
Kobe     Bean      Bryant
Elton    Hercules  John
Ben      Geza      Affleck
Matt     Paige     Damon
```

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;
}
```

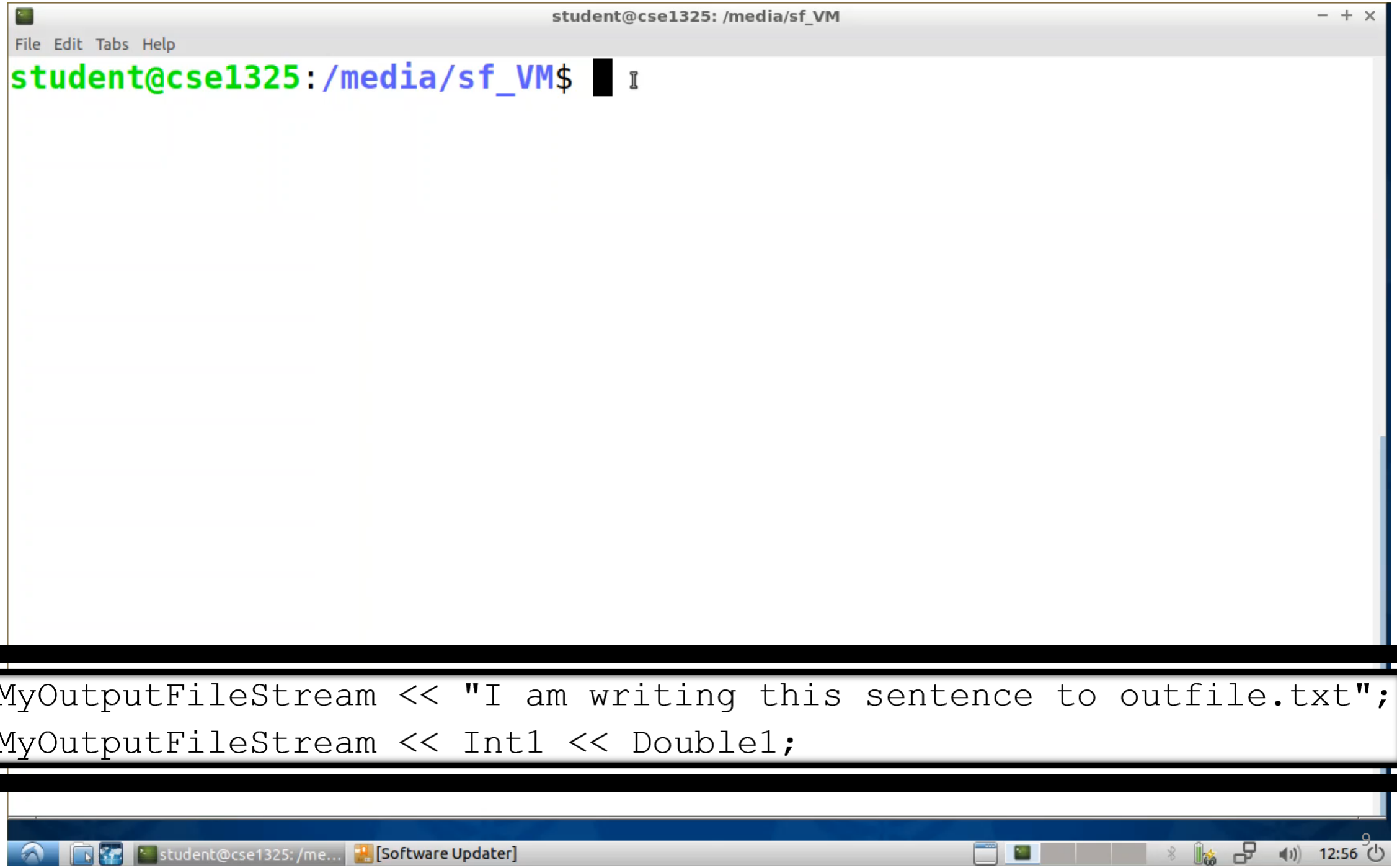
File Processing – Writing to a File

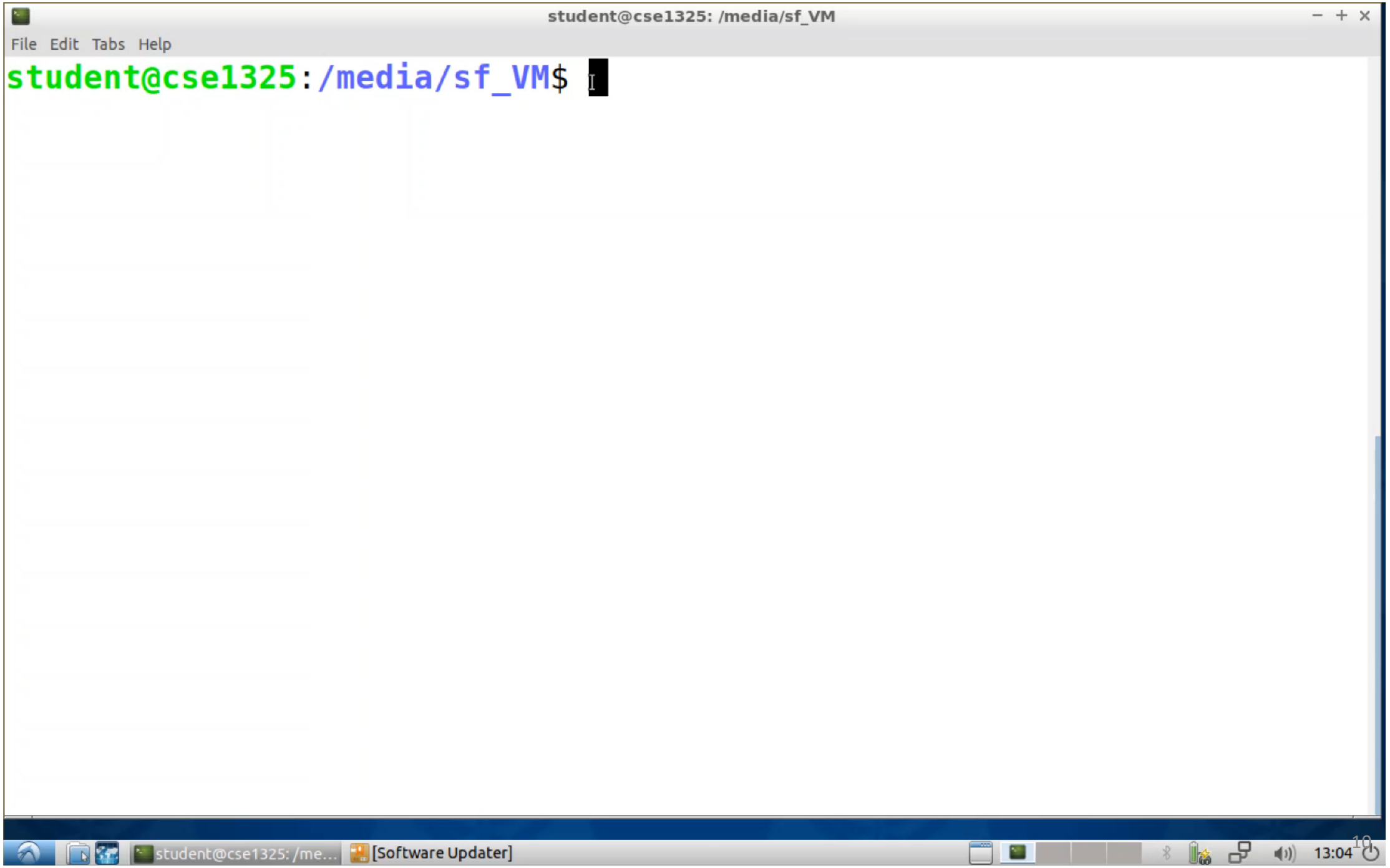
```
ofstream MyOutputFileStream{"outfile.txt", ios::out};
int Int1 = 10;
double Double1 = 12.34;

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

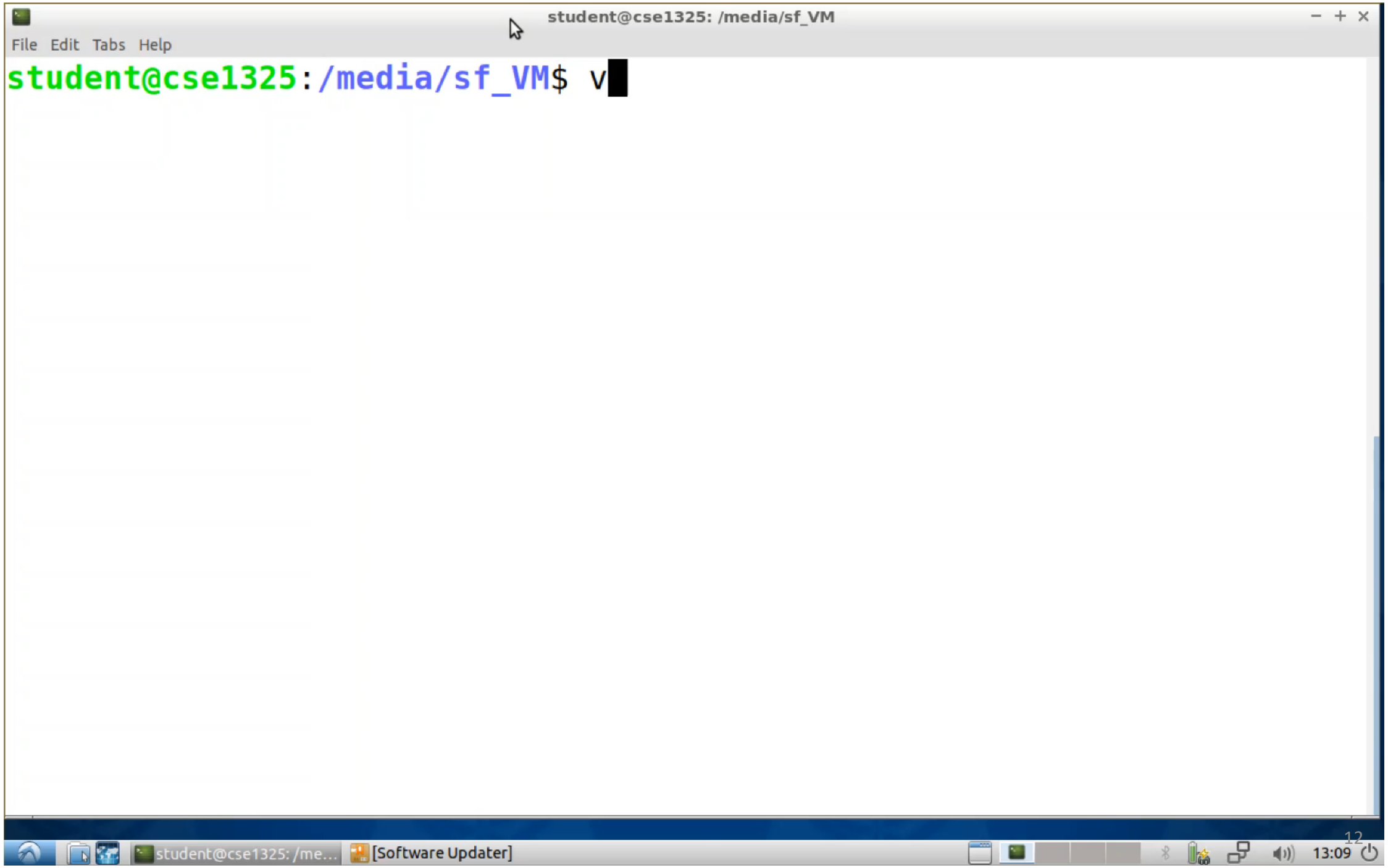
File Processing – Appending to a File

```
ofstream MyOutputFileStream{"outfile.txt", ios::app};
int Int1 = 100;
double Double1 = 120.34;

if (MyOutputFileStream.is_open())
{
   MyOutputFileStream << "\nWriting a new line to my existing file opened with append";
   MyOutputFileStream << endl << Int1 << endl << Double1;
}
else
{
   cout << "The file did not open" << endl;
}

MyOutputFileStream.close();
```

ofstream2Demo.cpp


student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ v
student@cse1325: /me...
[Software Updater]
13:09

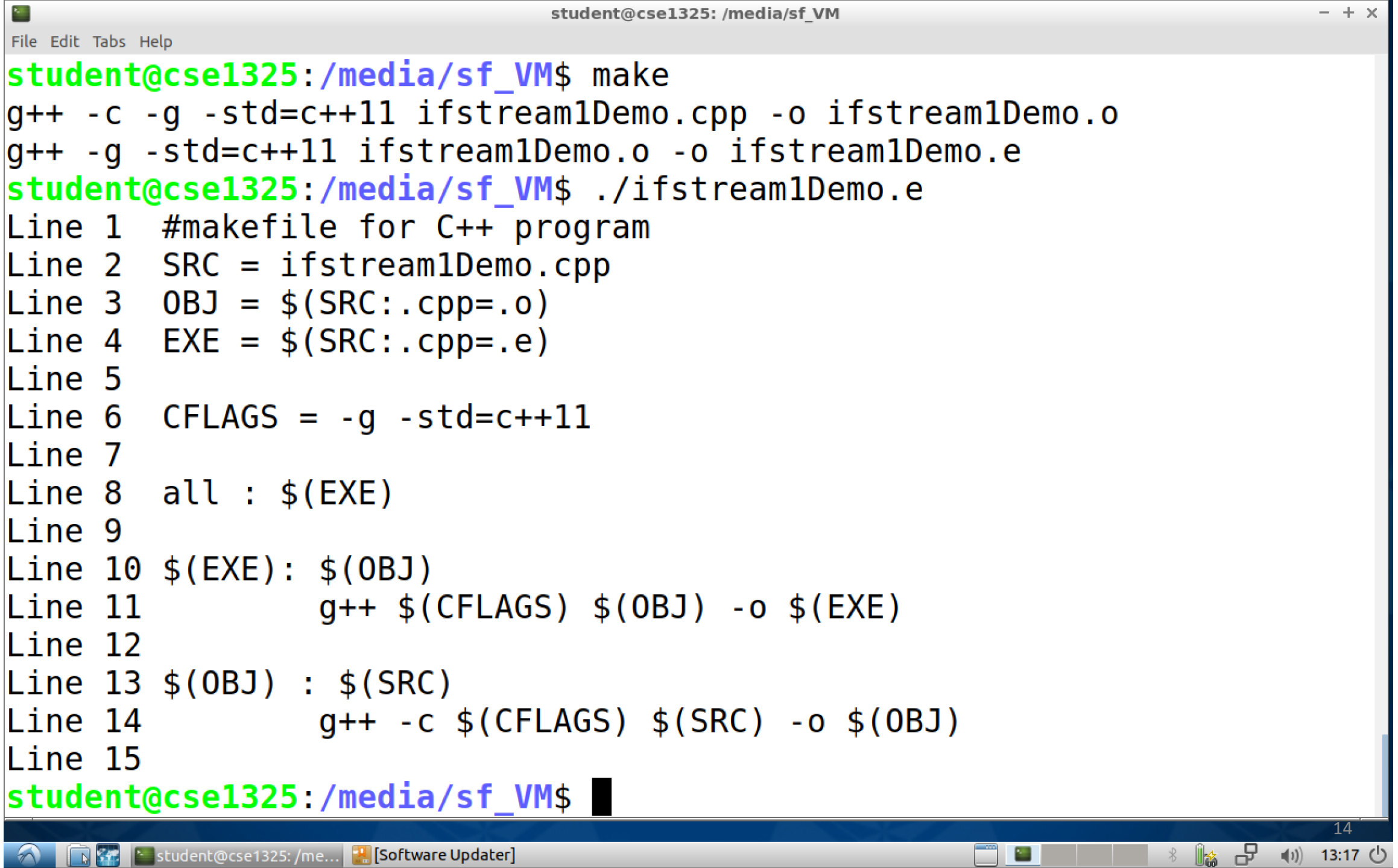
File Processing – Reading from a File

```
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 $(OBJ)
Line 15
student@cse1325:/media/sf_VM$
14
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.

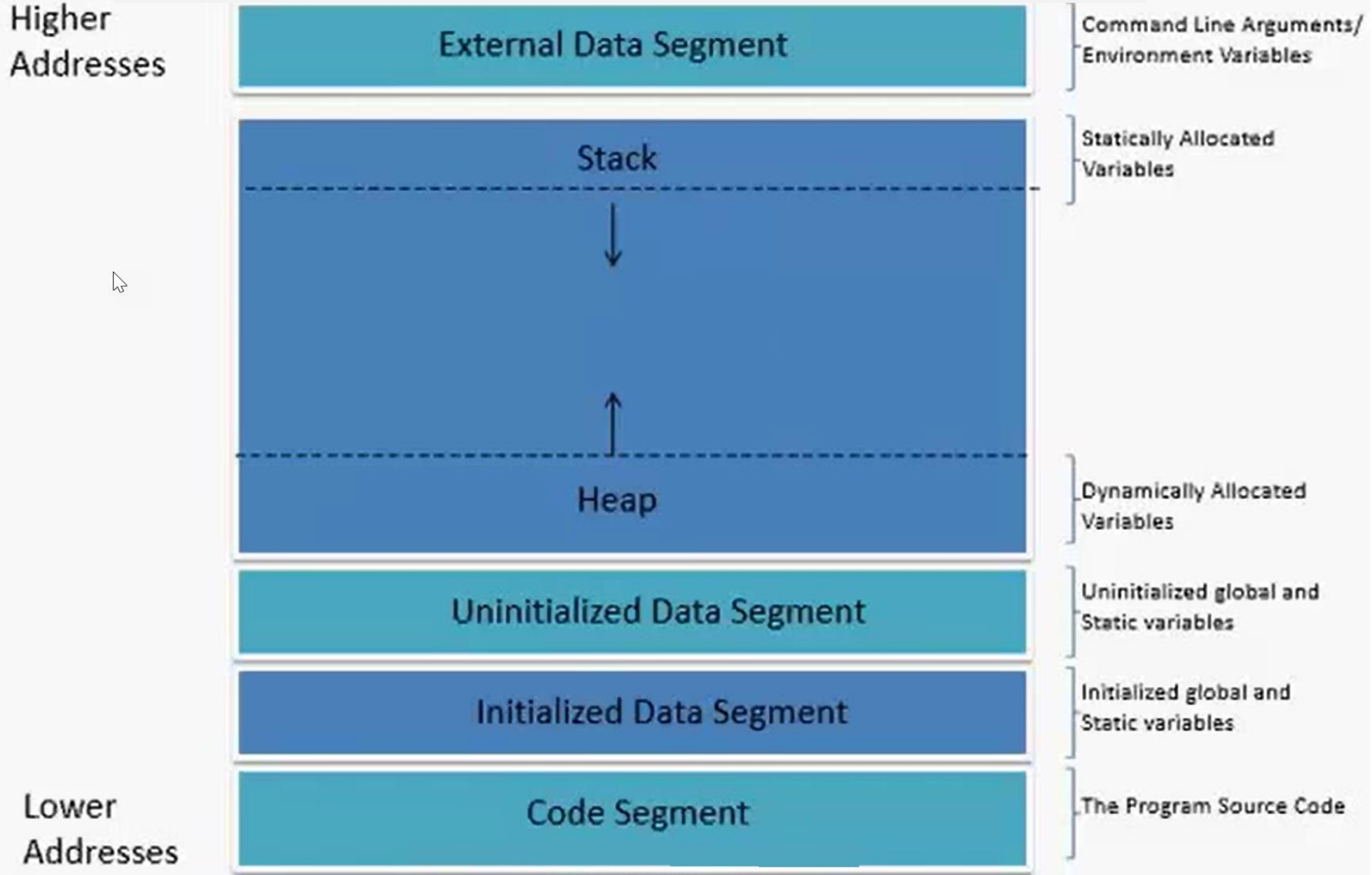
Dynamic Memory Allocation in C++

Dynamic memory allocation is a way for running programs to request memory from the operating system when needed.

This memory does not come from the program's limited stack memory.

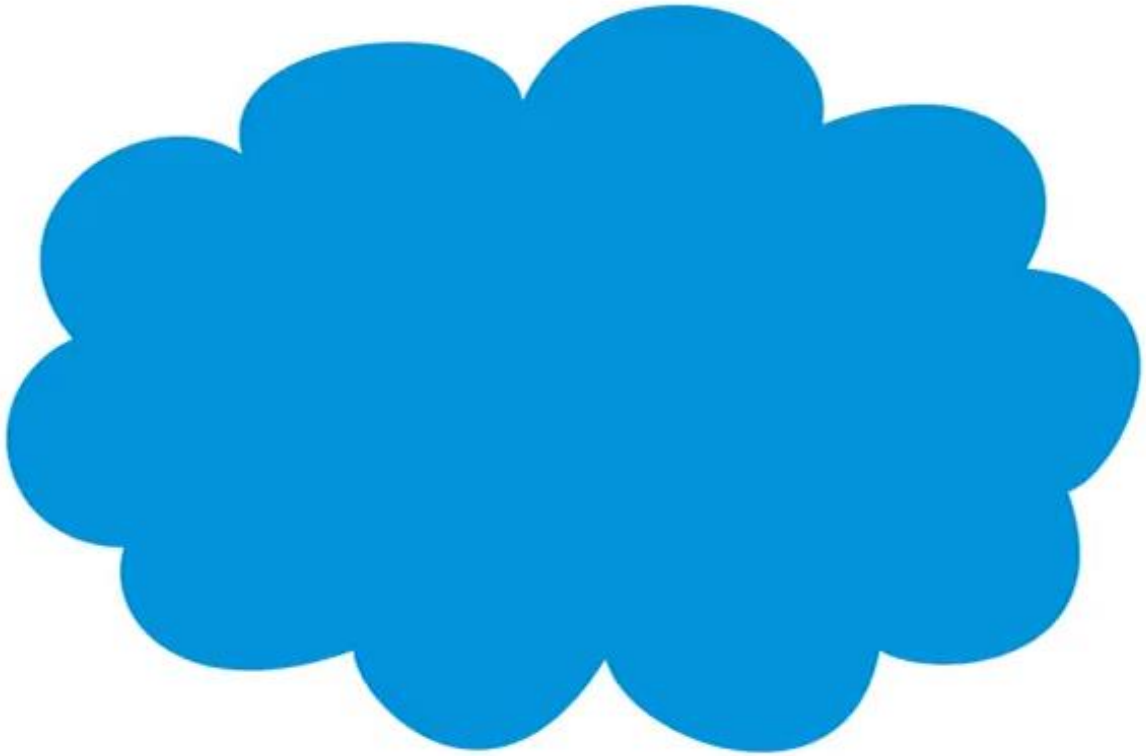
It is allocated from a much larger pool of memory managed by the operating system called the **heap**. On modern machines, the heap can be gigabytes in size.

Layout of Memory



Stack vs Heap

Stack



Heap

Heap Memory vs Stack Memory

- **Stack** is used for static memory allocation
  - static variables, strings, local variables, function parameters
  - faster than the heap
  - LIFO
  - do not deallocate variables
  - managed by CPU and will not become fragmented
  - has a predetermined size

- **Heap** is used for dynamic memory allocation
  - dynamically allocated variables
  - slower than the stack
  - random access
  - must free allocated memory
  - managed by programmer – can become fragmented
  - size only limited by machine’s memory

Both are stored in the computer's RAM .

`new`

You can control memory *allocation* and *deallocation* in C++ just like we did in C.

This is known as dynamic memory management

C used `malloc()`/`calloc()` and C++ uses `new`

C used `free()` and C++ uses `delete`

C used `realloc()` and C++

`new`

Anything created with `new` is created in the heap

> a region of memory assigned to each program for storing dynamically allocated objects.

Once memory is allocated, you can access it via the pointer that operator `new` returns.

Return memory by using the `delete` operator to deallocate it.

`new`

The `new` operator

- allocates storage of the proper size
- returns a **pointer** to the type specified to the right of the `new` operator.

If `new` is unable to find sufficient space in memory, it indicates that an error occurred by “throwing an exception.”

new

```
(gdb) p/a MyDynamic
$1 = 0x555555767e70
(gdb) p/a MyStatic
$2 = 0x7fff
(gdb) ptype MyStatic
type = int
(gdb) ptype MyDynamic
type = int *
```

To statically allocate memory for a variable

```
int MyStatic;
```

To dynamically allocate memory for a variable

```
int *MyDynamic  = new int;
```

Initializing Dynamic Memory

To initialize our static variable

```
MyStatic = 1;
```

```
(gdb) p MyStatic
$5 = 1
(gdb) p *MyDynamic
$6 = 1
```

To initialize our dynamic variable

```
*MyDynamic = 1;
```

Initializing Dynamic Memory

We can initialize a static variable with the declaration

```
int MyStatic{1};
```

```
(gdb) p MyStatic
$5 = 1
(gdb) p *MyDynamic
$6 = 1
```

We can do the same with a dynamic variable

```
int *MyDynamic{new int{1}}
```

Initializing Dynamic Memory

Write the statements to

- declare a dynamic variable named `Boat` of type `float` and initialize it during the declaration to `5.4`
- print `Boat`'s value

```
float *Boat{new float{5.4}};
std::cout << *Boat;
```

delete

To destroy/free dynamically allocated memory, use `delete`

```
delete MyDynamic;
```

This statement deallocates the memory by returning the memory to the heap.

delete

Do not delete memory that was not allocated by `new`. Doing so results in undefined behavior.

```
int MyStatic{1};
int *MyDynamic{new int{1}};

delete &MyStatic;
```

```
7                   int MyStatic{1};
(gdb)
8                   int *MyDynamic{new int{1}};
(gdb)
10                  delete &MyStatic;
(gdb) p &MyStatic
$1 = (int *) 0x7fffffffe07c
(gdb) step

Program received signal SIGSEGV, Segmentation fault.
```

`delete`

The `delete` operator does not *actually* delete anything.

It simply returns the memory being pointed to back to the operating system.

The operating system is then free to reassign that memory to another application (or to this application again later).

delete

```
7                   int MyStatic{1};
(gdb) step
8                   int *MyDynamic{new int{1}};
(gdb)
10                  delete MyDynamic;
(gdb) p MyDynamic
$1 = (int *) 0x555555767e70
(gdb) step
13                  return 0;
(gdb) p MyDynamic
$2 = (int *) 0x555555767e70
```

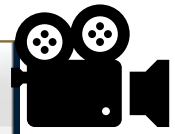
Memory Leak

Memory leaks happen when your program loses the address of dynamically allocated memory before giving it back to the operating system. Going out of the scope of a dynamically allocated variable is a good way to lose the address.

When this happens, your program cannot delete the dynamically allocated memory because it no longer knows where it is.

The operating system also cannot use this memory because that memory is considered to be still in use by your program.

Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well. Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash. Only after your program terminates is the operating system able to clean up and “reclaim” all leaked memory.



student@cse1325: /media/sf_VM

File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ gdb ./newDemo.e
```


student@cse1325: /me...

20:44

Null Pointer

After you delete a block of dynamically allocated memory, be sure not to delete the same block again.

One way to guard against this is to immediately set the pointer to `nullptr`.

Deleting a `nullptr` has no effect.

```
Breakpoint 1, main () at newDemo.cpp:7
7               int MyStatic{1};
(gdb) step
8               int *MyDynamic{new int{1}};
(gdb)
10              delete MyDynamic;
(gdb)
11              MyDynamic = nullptr;
(gdb) p MyDynamic
$1 = (int *) 0x555555767e70
(gdb) step
14              return 0;
(gdb) p MyDynamic
$2 = (int *) 0x0
```

C++ Standard Library

C++ programs are typically written by combining “prepackaged” functions and classes available in the C++ Standard Library with new functions and classes you write.

The C++ Standard Library provides a rich collection of functions.

There are three key components of the Standard Library—containers (*templatized* data structures), iterators and algorithms.

C++ Standard Library

Containers are data structures capable of storing objects of almost any data type (there are some restrictions).

We'll see that there are three styles of container classes—first-class containers, container adapters and near containers.

The containers are divided into four major categories—sequence containers, ordered associative containers, unordered associative containers and container adapters.

C++ Standard Library

Container class	Description
Unordered associative containers	
`unordered_set`	Rapid lookup, no duplicates allowed.
`unordered_multiset`	Rapid lookup, duplicates allowed.
`unordered_map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`unordered_multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.

C++ Standard Library

Container class	Description
Sequence containers	
`array`	Fixed size. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`forward_list`	Singly linked list, rapid insertion and deletion anywhere. Added in C++11.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
`vector`	Rapid insertions and deletions at back. Direct access to any element.

Sequence Containers

arrays

fixed-size collections consisting of data items of the same type

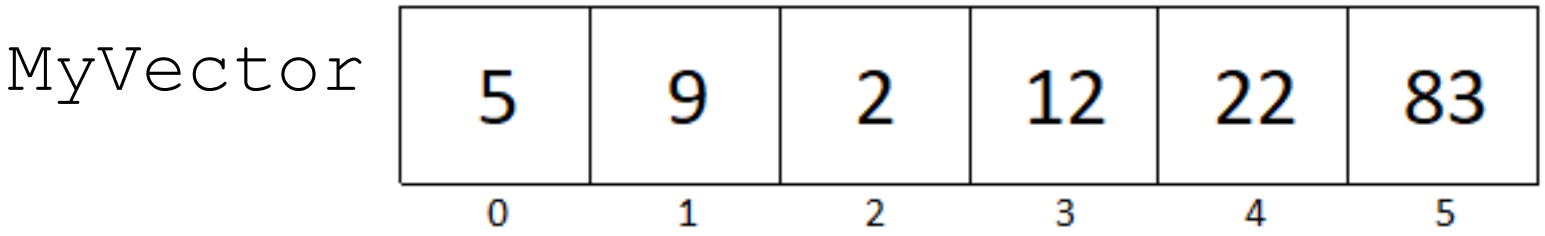
vectors

collections consisting of data items of the same type that can grow and shrink dynamically at execution time.

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

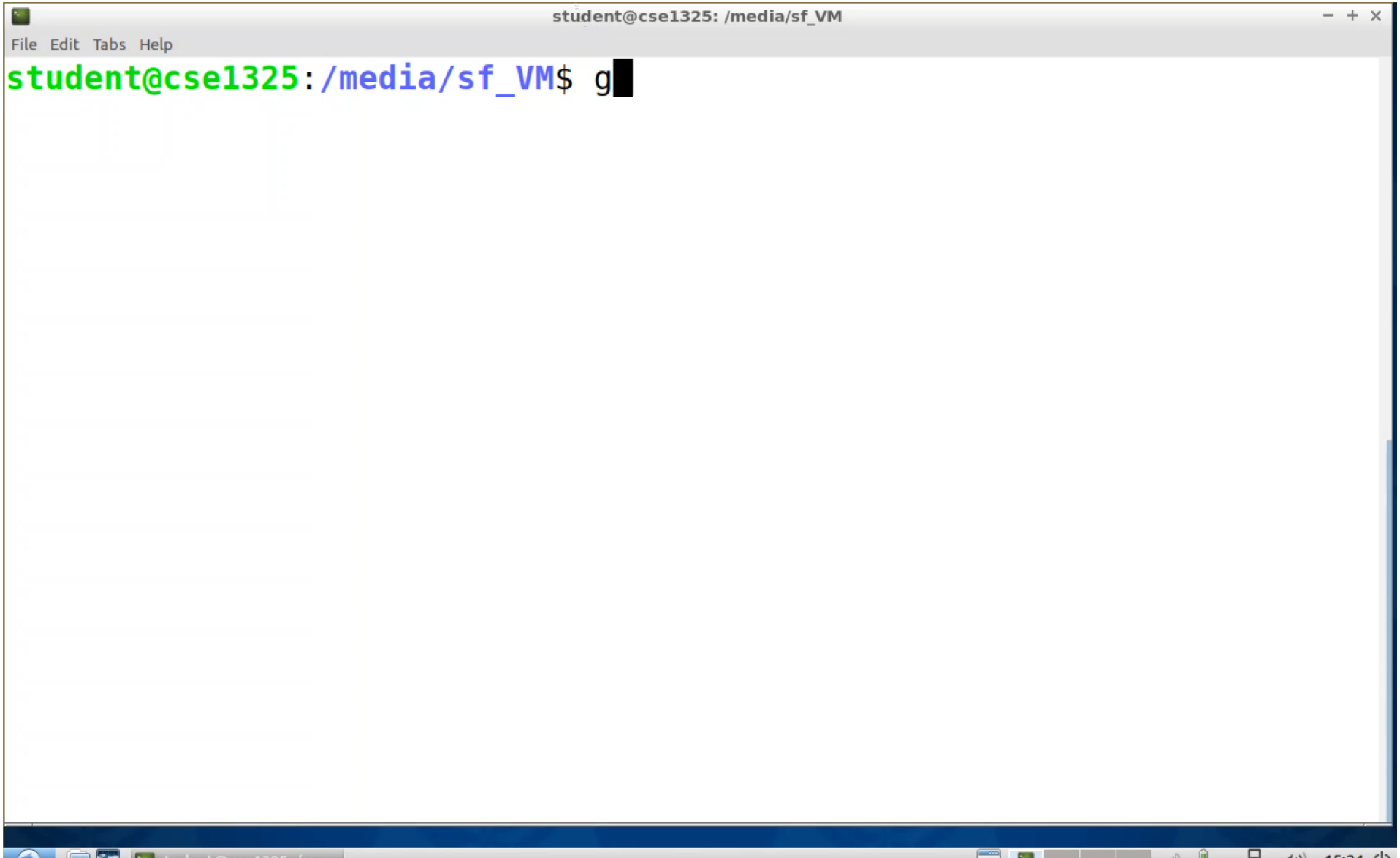
```
#include <vector>
```

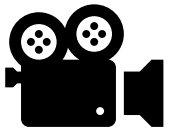
- Declaring a vector

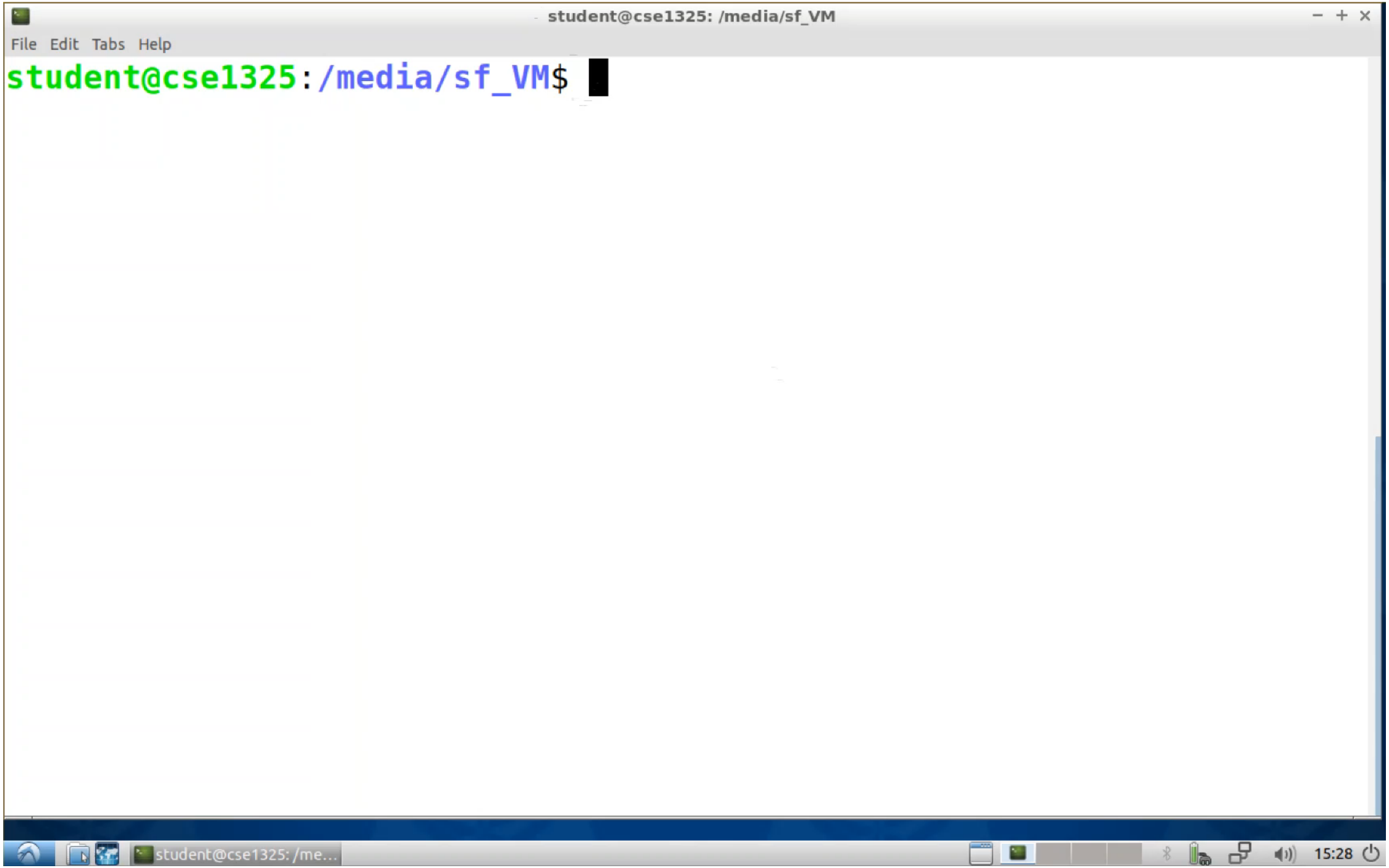
```
vector<type> vectorname;
vector<type> vectorname(number of elements);
```

- Initializing and declaring a vector

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


student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ g
student@cse1325: /me...
15:24




student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$
student@cse1325: /me...
15:28

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

The range-based for statement can be used in place of the counter-controlled for statement whenever code looping through a vector does not require access to the element's subscript.

If a program needs use subscripts for some reason other than simply to loop through a vector

to print a subscript number next to each array element value

use the counter-controlled for statement.

vector

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

for (i = 0; i < Bank.size(); i++)
    cout << setprecision(i) << setw(i+2) << Bank[i];

 1  2 3.5 4.57 5.679
```

How do we change this to be a range based for statement?

```
student@cse1325:/media/sf_VM$ ./vector5Demo.e
You are in
                                                  CSE1325

setw() is not sticky
student@cse1325:/media/sf_VM$
```

```
vector <string> ClassName{"CSE1325"};

cout << "You are in " << endl;

for (string it : ClassName)
    cout << setw(50) << it;

cout << "\nsetw() is not sticky" << endl;
```

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

vector

```
#include <iostream>

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

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

    return 0;
}
```

```
student@cse1325:/media/sf_VM$ make
make: Warning: File 'makefile' has modification time 137 s in the futu
re
g++ -c -g -std=c++11 auto1Demo.cpp -o auto1Demo.o
auto1Demo.cpp: In function ‘int main()’:
auto1Demo.cpp:7:2: error: ‘vector’ was not declared in this scope
  vector<string>CatNames{"Shade", "Appa", "Sylvester", "Josie"};
  ^~~~~~
auto1Demo.cpp:7:2: note: suggested alternative: ‘perror’
  vector<string>CatNames{"Shade", "Appa", "Sylvester", "Josie"};
  ^~~~~~
  perror
auto1Demo.cpp:7:9: error: ‘string’ was not declared in this scope
  vector<string>CatNames{"Shade", "Appa", "Sylvester", "Josie"};
         ^~~~~~
auto1Demo.cpp:7:9: note: suggested alternatives:
In file included from /usr/include/c++/7/iosfwd:39:0,
                 from /usr/include/c++/7/ios:38,
                 from /usr/include/c++/7/ostream:38,
                 from /usr/include/c++/7/iostream:39,
```

```
#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)
      2     4     6     8     10                            cout << x << "\t";

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

MyVector after push_back(12)

      2     4     6     8     10    12

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

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

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

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

```
    2    4    6    8
MyVector.size() 4 MyVector.capacity() 4
```

capacity()

Vectors may allocate extra capacity

When a vector is resized, the vector may allocate more capacity than is needed.

This is done to provide some “breathing room” for additional elements, to minimize the number of resize operations needed.

Allows the vector to not need to reallocate every time a push_back is done.