CSE 1325

Week of 09/14/2020

Instructor : Donna French

The `auto` keyword

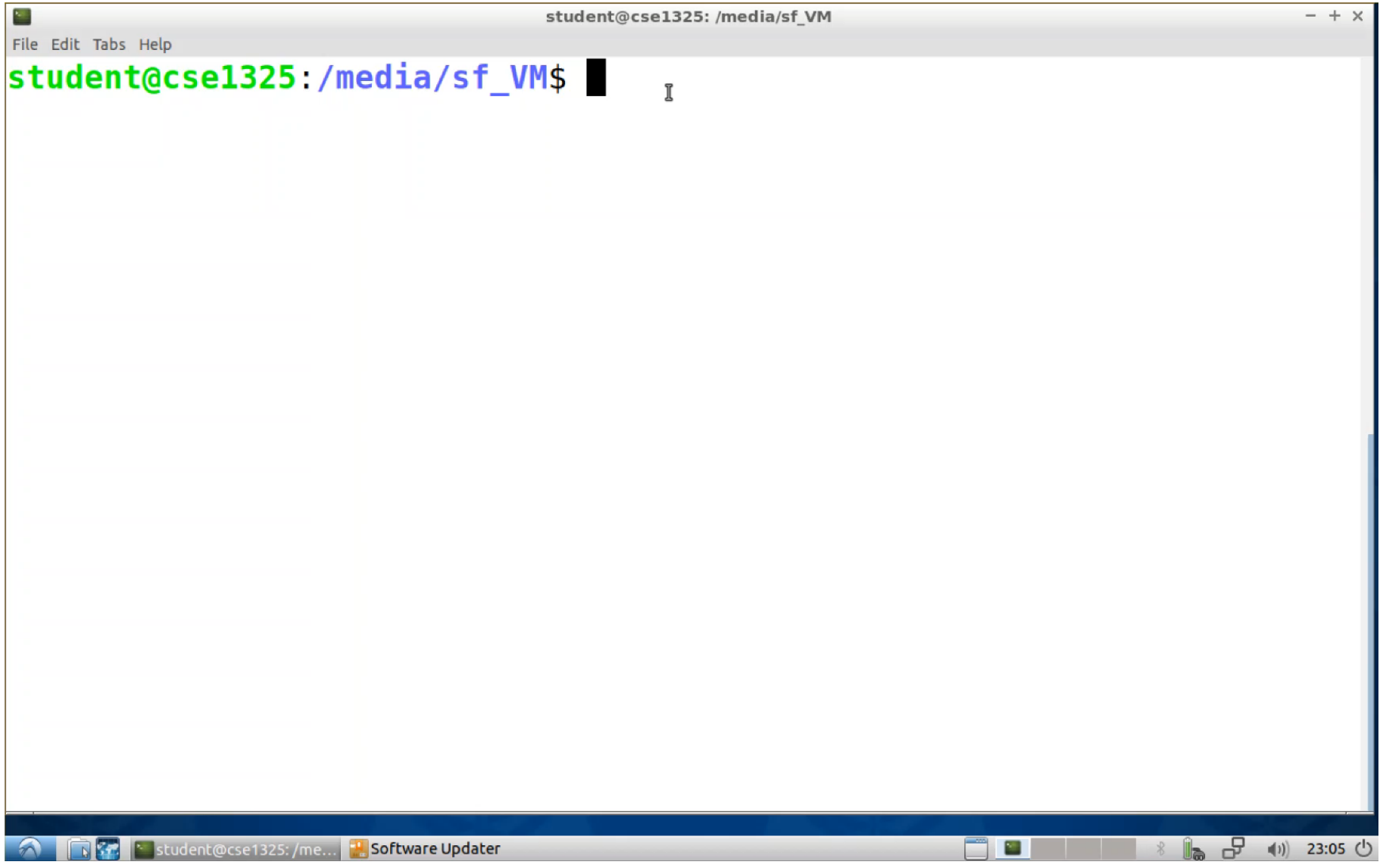
When initializing a variable, the `auto` keyword can be used in place of the variable type to tell the compiler to infer the variable's type from the initializer's type.

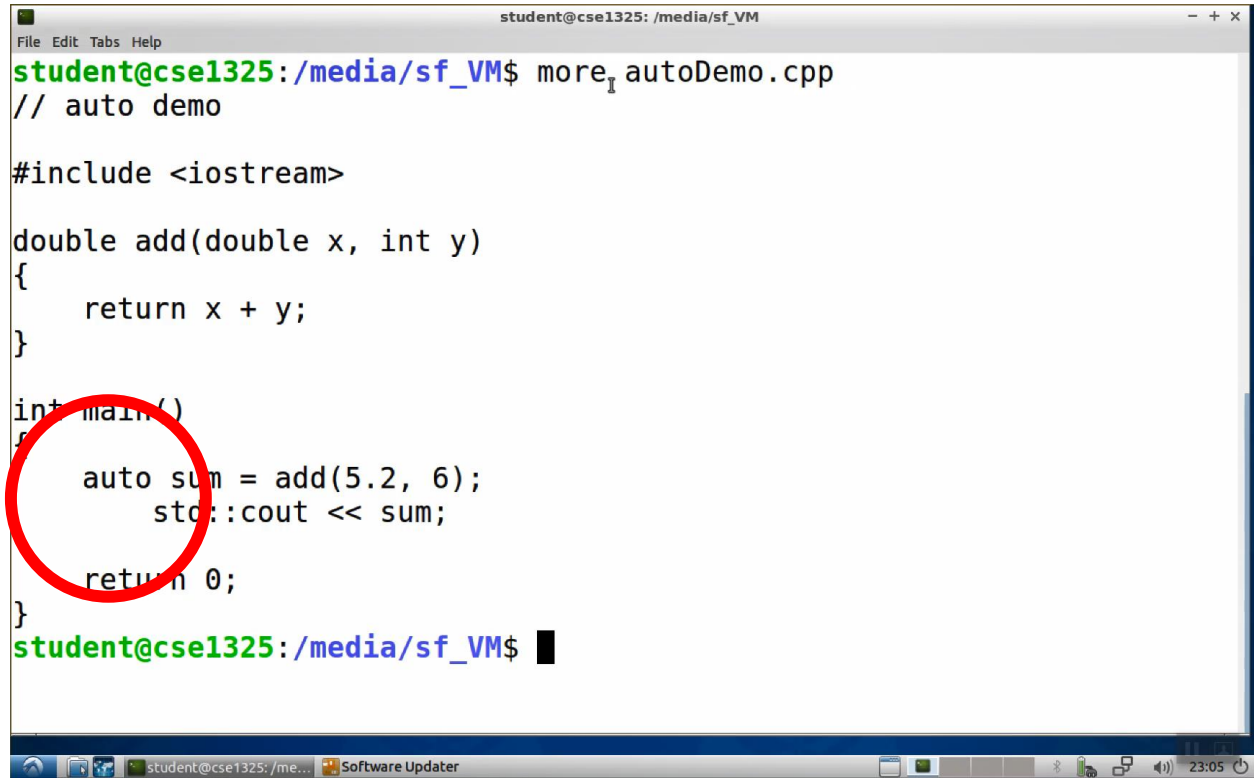
This is called **type inference** (also sometimes called type deduction).

```
auto d = 5.0;
auto i = 1 + 2;
```

The `auto` keyword

This even works with the return values from functions






```
1  // auto demo
2
3  #include <iostream>
4
5  double add(double x, int y)
6  {
7      return x + y;
8  }
9
10 int main()
11 {
12     auto sum = add(5.2, 6);
13     std::cout << sum;
14
15     return 0;
16 }
```

The `auto` keyword

- only works when initializing a variable upon creation. Variables created without initialization values cannot use this feature (as C++ has no context from which to deduce the type).
- the compiler cannot infer types for function parameters at compile time; therefore, `auto` cannot be used for function parameters
- best used when the object's type is hard to type, but the type is obvious from the right hand side of the expression
- using `auto` in place of fundamental data types only saves a few (if any) keystrokes – in the future, we will see examples where the types get complex and lengthy. In those cases, using `auto` can be very nice.

Familiar C++ Libraries

Should look very familiar

```
atoi(), atof()
    #include <cstdlib>

isdigit(), isalpha(), isalnum(), islower(),
isupper(), isspace(), ispunct()
    #include <ctype>
```

famlibDemo.cpp

```
char MyChar;
char MyCharNumber[10];
int MyInt;
float MyFloat;

cout << "Enter a number to be stored in MyCharNumber ";
cin >> MyCharNumber;

MyFloat = atof(MyCharNumber);
cout << "MyCharNumber is " << MyCharNumber << "\tThe float conversion is " << MyFloat << endl;

MyInt = atoi(MyCharNumber);
cout << "MyCharNumber is " << MyCharNumber << "\tThe int conversion is " << MyInt << endl;

cout << "\n\n\nEnter a letter to be UPPERCASED ";
cin >> MyChar;

MyChar = toupper(MyChar);

cout << MyChar << endl;
```

Unary Scope Resolution Operator

C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope.

The unary scope resolution operator (::) cannot be used to access a local variable of the same name in an outer block.

A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

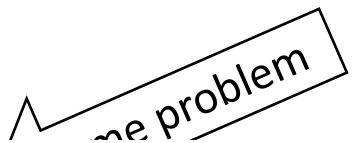
Unary Scope Resolution Operator

Global overridden by local in C

```
/* Global version of X */
int X = 0;

int main(void)
{
  /* Local version of X */
  int X = 123;

  printf("X = %d\n", X);

  return 0;
}

X = 123
```

Global overridden by local in C++

```
/* Global version of X */
int X = 0;

int main(void)
{
   /* Local version of X */
   int X = 123;

   cout << "X = " << ::X << endl;

   return 0;
}

X = 0
```

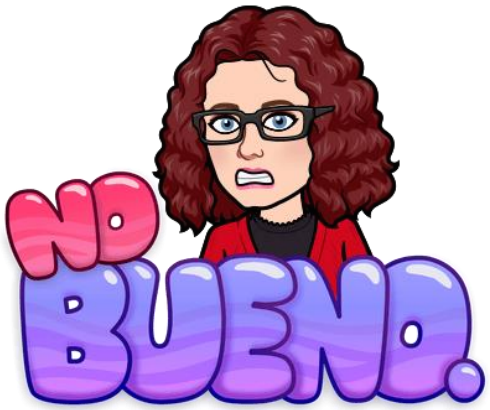


Programmer Joke

What is the best prefix for a global variable?

//

Global variables are not allowed in Coding Assignments unless specifically included as part of an assignment.



Streams

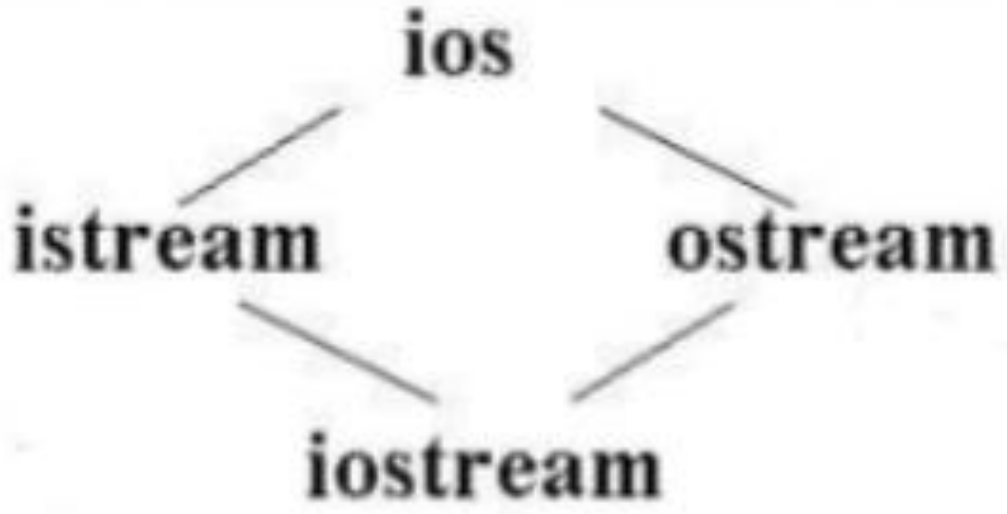
- C++ I/O occurs in streams of bytes
- A stream is a sequence of bytes
  - Input – bytes flow from a device (e.g., keyboard, drive) to memory
  - Output – bytes flow from memory to a device (e.g., screen, printer)
- C++ provides
  - low-level I/O capabilities
    - unformatted
    - high speed and high 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_????
```


Terminal - student@maverick: /media/sf_VM/CSE1325
File Edit View Terminal Tabs Help
student@maverick:/media/sf_VM/CSE1325$ gdb ./PrintAddress1Demo.e

Streams
Input/Output Member Functions

`cin.get()`

`get` is a member function of `cin`

retrieves a single character from the standard input stream

returns `EOF` when the end of file on the stream is encountered

`cout.put()`

`put` is a member function of `cout`

puts one character to the standard output stream

```c
#include <stdio.h>

int main(void)
{
   int c;

   while ((c = getchar()) != EOF)
   {
      putchar(c);
   }

   return 0;
}
```

```cpp
#include <iostream>

using namespace std;

int main()
{
   int c;

   while ((c = cin.get()) != EOF)
   {
      cout.put(c);
   }

   return 0;
}
```

getputDemo.cpp

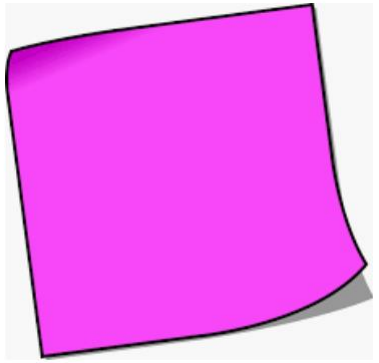
Stream Manipulators

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

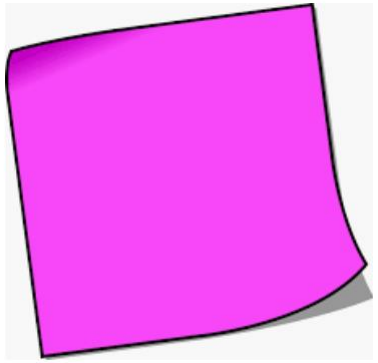
Sticky vs. Non-Sticky Stream Manipulators

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";
```

C++ manipulator showbase() function is used to set the showbase format flag for the str stream.

```
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)`

showbase & noshowbase are just flags to enable and disable the program to allow the values to be displayed in various types. Even after noshowbase is used if we use the words such as hex,dec,etc., there is no difference and the original values will be displayed

showbase is STICKY while setbase is not STiCKY

`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 Manipulators
Floating Point

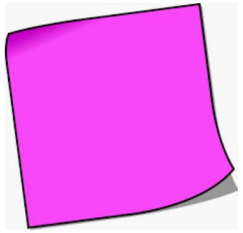
`setprecision,precision`

The precision (number of digits displayed) can be controlled for floating point numbers.

`precision`

member function of `cout`

`setprecision`

stream manipulator 

precision is reset by capturing current value before altering it and then passing that value to `cout.precision()` to reset the stream

```
double SR1 = sqrt(82.0);
streamsize MyStreamSize = cout.precision();
```
Means to tell to capture the precision

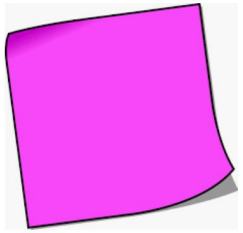
```
cout << "Square root of 82 with no precision \t\t" << SR1
     << "\n\n" << endl;

for (int i = 1; i < 10; i++)
{
     cout << "Square root of 82 with a precision of " << i
          << "\t\t" << setprecision(i) << SR1 << endl;
}

cout.precision(MyStreamSize);
```
Setting back to the captured precision

```
cout << "\n\nSquare root of 82 with original precision reset "
     << "\t" << SR1 << endl;
```

iomanipDemo.cpp

```
Square root of 82 with no precision                 9.05539


Square root of 82 with a precision of 1             9
Square root of 82 with a precision of 2             9.1
Square root of 82 with a precision of 3             9.06
Square root of 82 with a precision of 4             9.055
Square root of 82 with a precision of 5             9.0554
Square root of 82 with a precision of 6             9.05539
Square root of 82 with a precision of 7             9.055385
Square root of 82 with a precision of 8             9.0553851
Square root of 82 with a precision of 9             9.05538514


Square root of 82 with original precision reset         9.05539
```

Stream Manipulation
Floating Point



`fixed`, `scientific`

Used to control the output format of floating point numbers

`scientific`

forces a floating point number to display in scientific notation

`fixed`

forces a floating point number to display a specific number of digits to the right of the decimal

both of these change the stream – use `defaultfloat` to reset to the default

```
double SR2 = sqrt(82.0);

cout << "Square root of 82 with no stream notation set\t\t"
     << SR2 << "\n\n" << endl;
cout << "Square root of 82 in scientific notation     \t\t" << scientific
     << SR2 << "\n\n" << endl;
cout << "Square root of 82 with no stream notation set\t\t"
     << SR2 << "\n\n" << endl;

cout << "Square root of 82 in fixed notation          \t\t" << fixed
     << SR2 << "\n\n" << endl;
cout << "Square root of 82 with no stream notation set\t\t"
     << SR2 << "\n\n" << endl;
cout << defaultfloat;
cout << "Square root of 82 after resetting to default \t\t"
     << SR2 << "\n\n" << endl;
```

iomanipDemo.cpp

```
Square root of 82 with no stream notation set          9.05539

Square root of 82 in scientific notation               9.055385e+00

Square root of 82 with no stream notation set          9.055385e+00

Square root of 82 in fixed notation                    9.055385

Square root of 82 with no stream notation set          9.055385

Square root of 82 after resetting to default           9.05539
```

Stream Manipulators
Field Width

`setw, width`

Controls the width - number of character positions in which a value should be output – right justifies text

`width`

member function of `cout`

sets the width for the next `cout`

`setw`

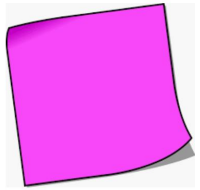
stream manipulator

```
string MyString = "CSE1325";

cout << "Printed with no width specified---" << MyString << endl;
cout.width(40);
cout << "Printed with width set to 40---" << MyString << endl;

for (int i = 10; i < 20; i++)
{
      cout << "Printed with a width of " << i << "----"
           << setw(i) << MyString << endl;
}

cout << "\nPrinted with no width specified---" << MyString << endl;
```

iomanipDemo.cpp

```
Printed without width set------CSE1325

         1         2         3         4         5
12345678901234567890123456789012345678901234567890

         Printed with width set to 40---CSE1325

         1         2         3         4         5
12345678901234567890123456789012345678901234567890

Printed with a width of 10----   CSE1325
Printed with a width of 11----    CSE1325
Printed with a width of 12----     CSE1325
Printed with a width of 13----      CSE1325
Printed with a width of 14----       CSE1325
Printed with a width of 15----        CSE1325
Printed with a width of 16----         CSE1325
Printed with a width of 17----          CSE1325
Printed with a width of 18----           CSE1325
Printed with a width of 19----            CSE1325

Printed with no width specified---CSE1325
```

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

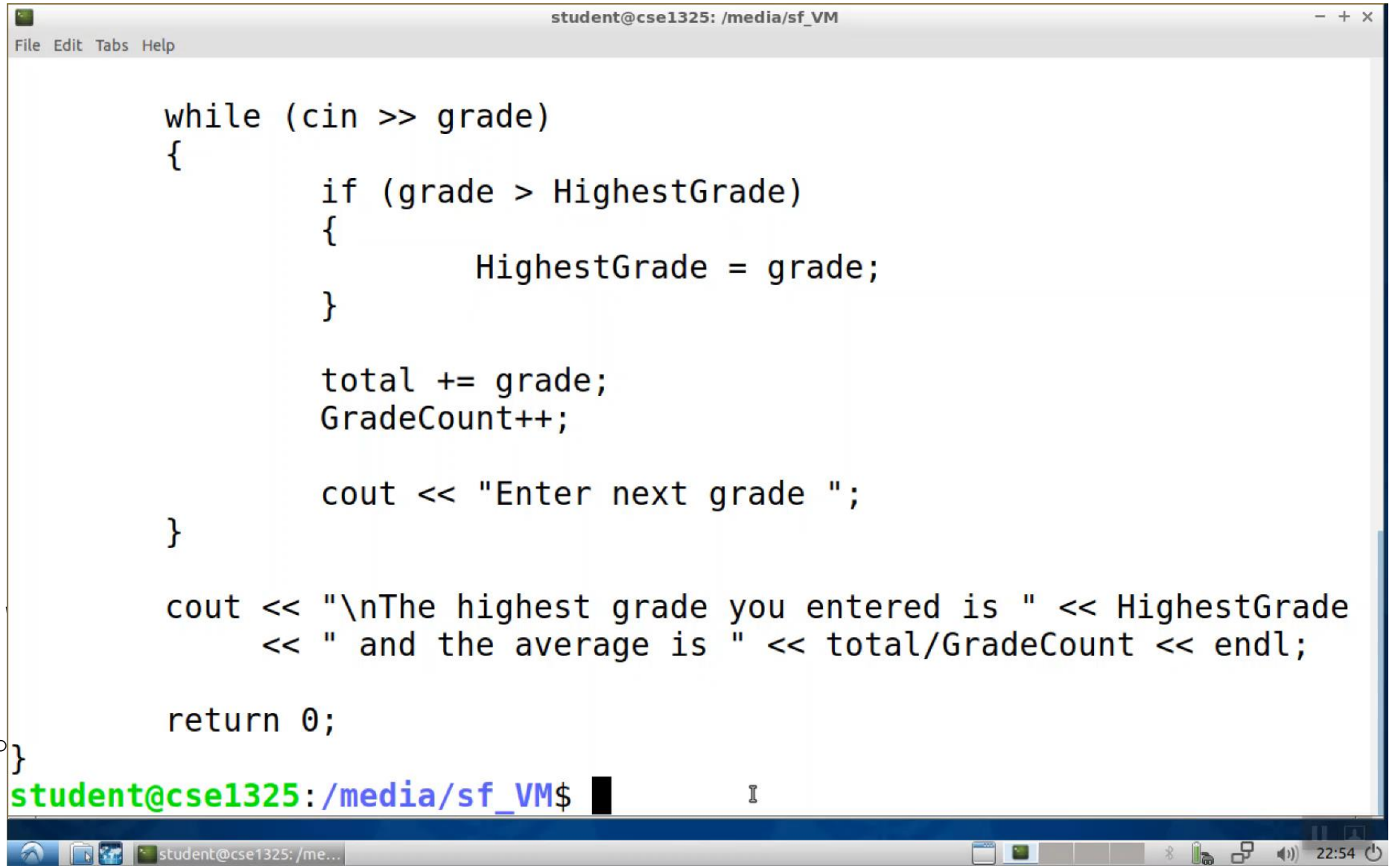
Using `cin` as the condition of a `while` loop

```
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

```
#include <iostream>

using namespace std;

int main()
{
   string MySentence;

   cout << "Enter a sentence ";

   while (cin >> MySentence)
   {
      cout << MySentence << endl;
   }
   return 0;
}
```

whilecout2Demo.cpp

Does entering a number cause the `cin` `while` to stop?

No, because a number is treated like a character.

So if letters and numbers are accepted by `cin` into the string `MySentence`, then how to make this loop quit?

Ctrl-D causes `cin` to return 0 which makes the `while` loop quit.

`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.

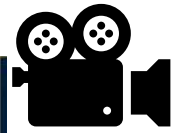
When the user enters input in response to an extraction operation, that data is placed in a buffer.

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.

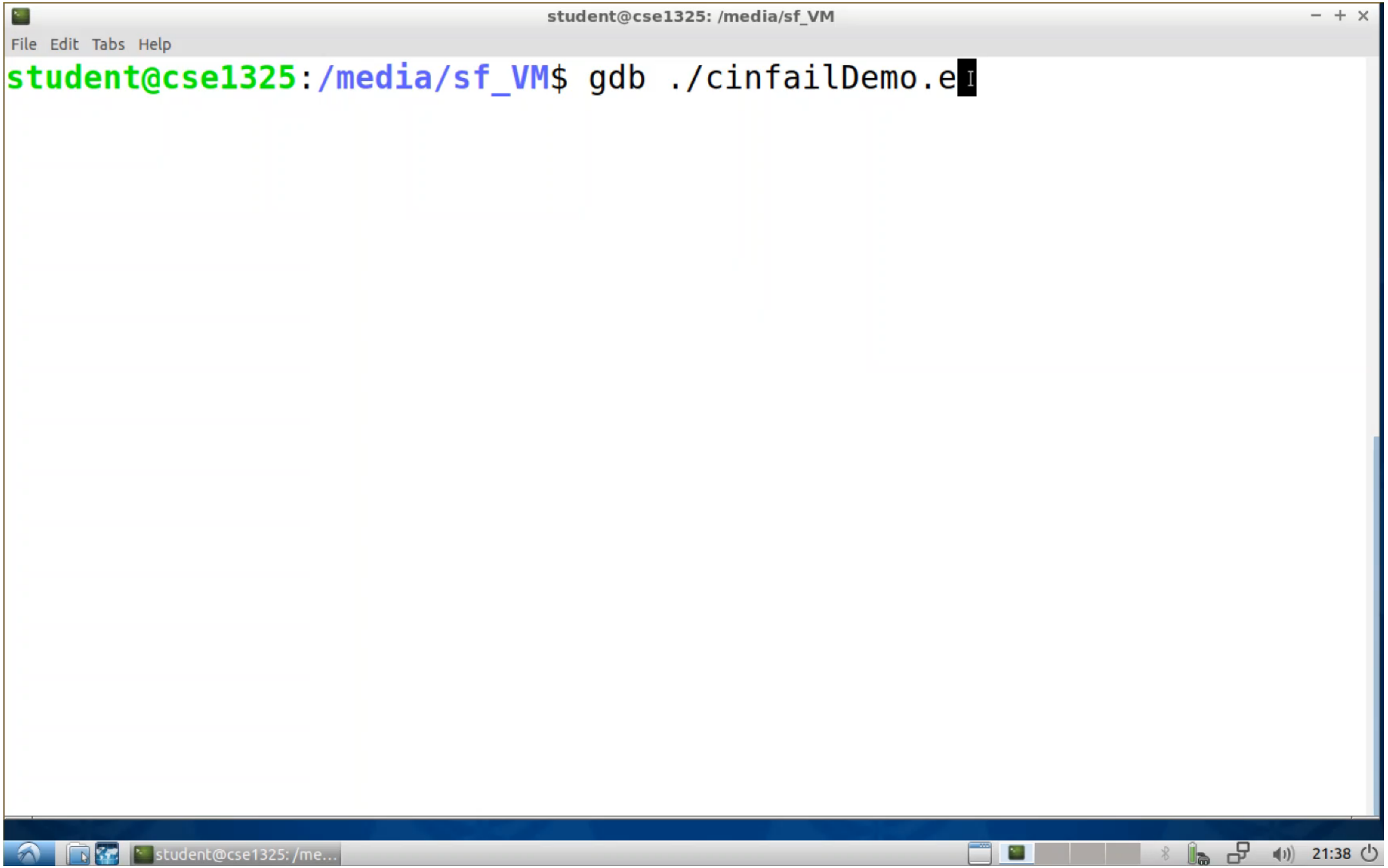


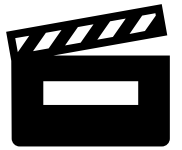
student@cse1325: /media/sf_VM

File Edit Tabs Help

```
#include <iostream>

using namespace std;

int main()
{
        int x = 0, y = 0;

        cout << "Please the first integer ";
        cin >> x;
        cout << "You entered " << x << endl;

        cout << "Please enter the second integer ";
        cin >> y;
        cout << "You entered " << y << endl;

        return 0;
}
```

```
student@cse1325:/media/sf_VM$
```


student@cse1325: /media/sf_VM
File Edit Tabs Help
student@cse1325:/media/sf_VM$ gdb ./cinfailDemo.e
student@cse1325: /me...
21:38

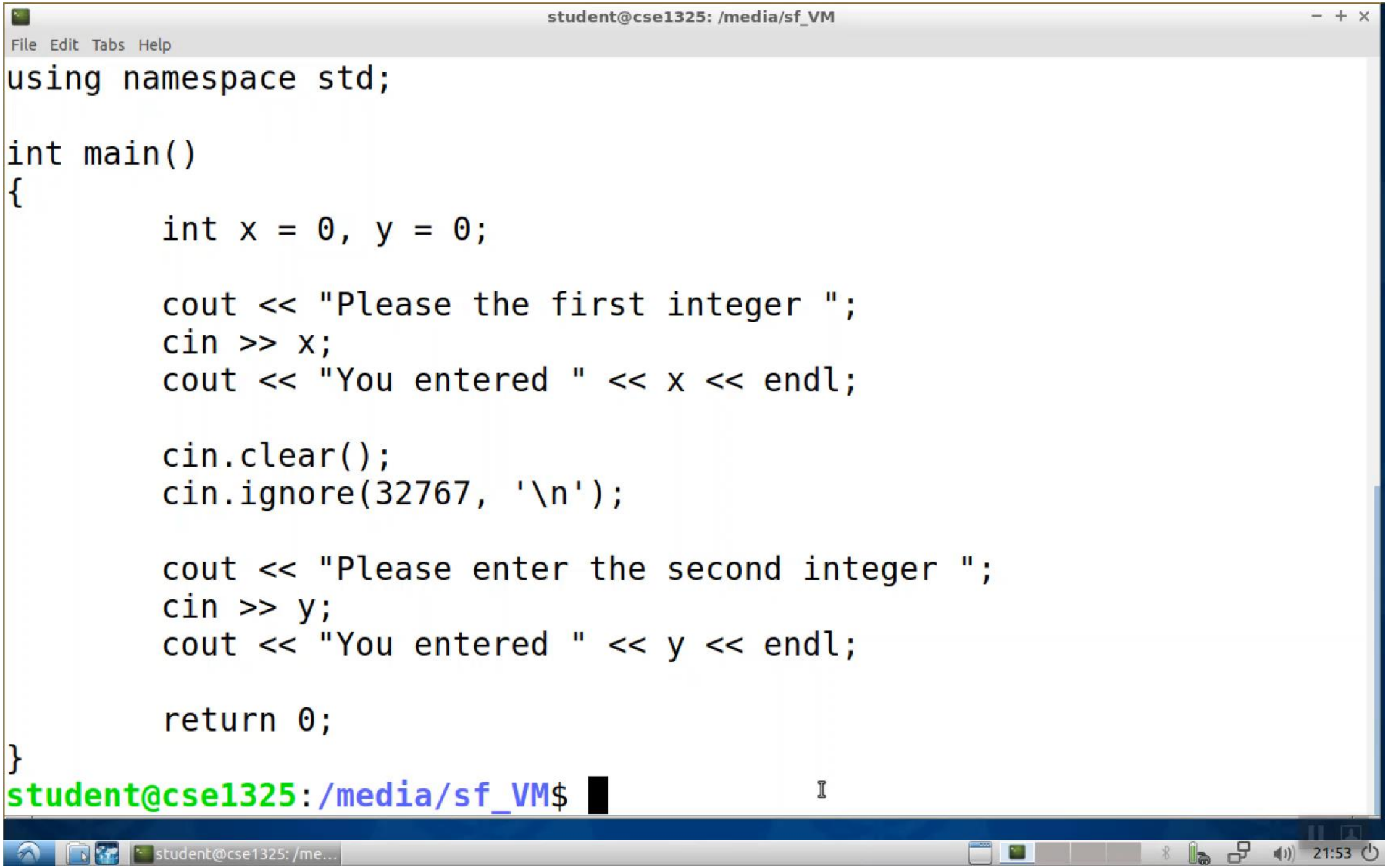


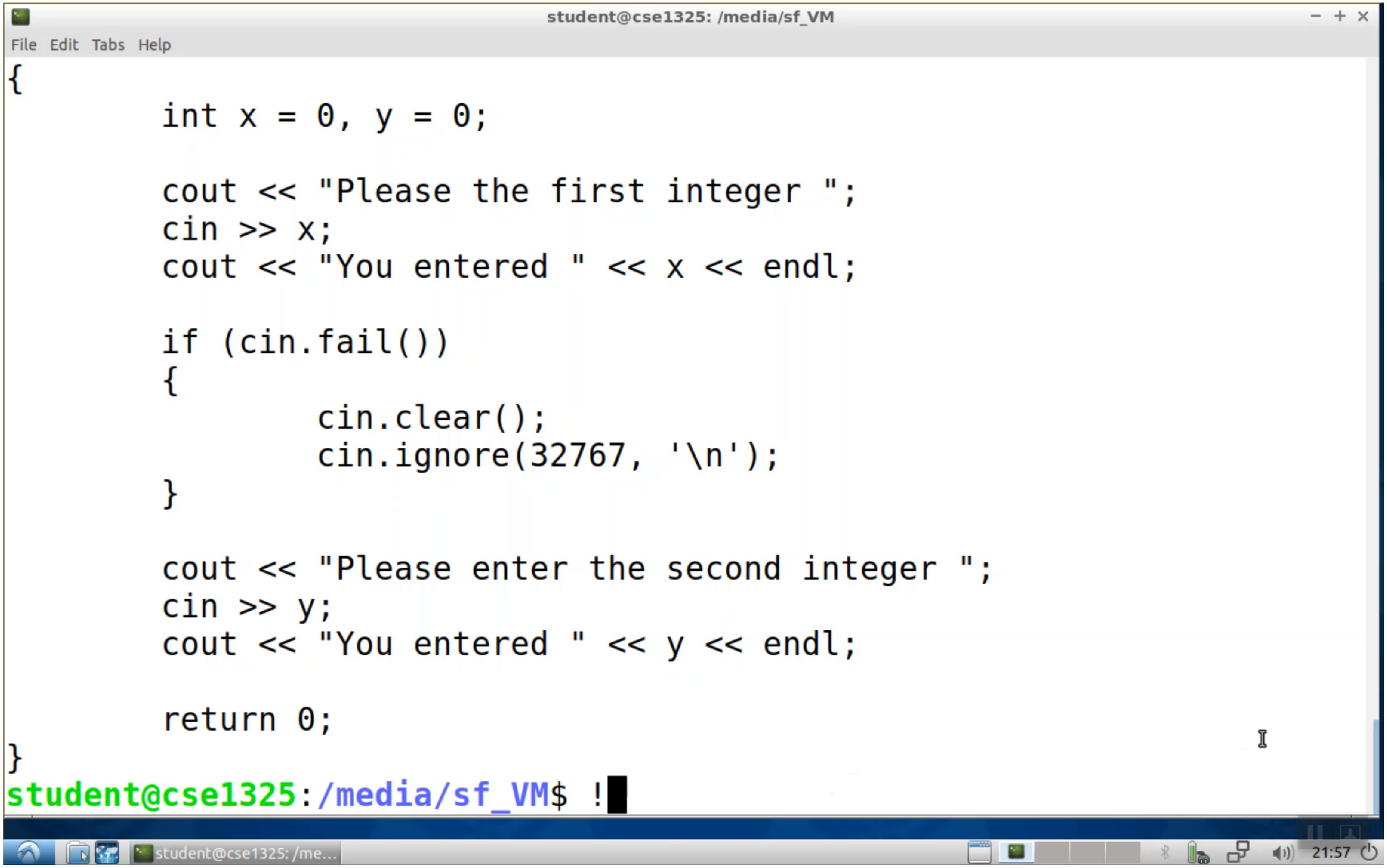
student@cse1325: /media/sf_VM

File Edit Tabs Help

```
using namespace std;

int main()
{
        int x = 0, y = 0;

        cout << "Please the first integer ";
        cin >> x;
        cout << "You entered " << x << endl;

        cin.ignore(32767, '\n');

        cout << "Please enter the second integer ";
        cin >> y;
        cout << "You entered " << y << endl;

        return 0;
}
```

```
student@cse1325:/media/sf_VM$
```


student@cse1325: /media/sf_VM
File Edit Tabs Help
using namespace std;
int main()
{
int x = 0, y = 0;
cout << "Please the first integer ";
cin >> x;
cout << "You entered " << x << endl;
cin.clear();
cin.ignore(32767, '\n');
cout << "Please enter the second integer ";
cin >> y;
cout << "You entered " << y << endl;
return 0;
}
student@cse1325:/media/sf_VM$
student@cse1325: /me...
21:53


student@cse1325: /media/sf_VM
File Edit Tabs Help
{
int x = 0, y = 0;
cout << "Please the first integer ";
cin >> x;
cout << "You entered " << x << endl;
if (cin.fail())
{
cin.clear();
cin.ignore(32767, '\n');
}
cout << "Please enter the second integer ";
cin >> y;
cout << "You entered " << y << endl;
return 0;
}
student@cse1325:/media/sf_VM$!
student@cse1325: /me...
21:57

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

Happy Path Testing

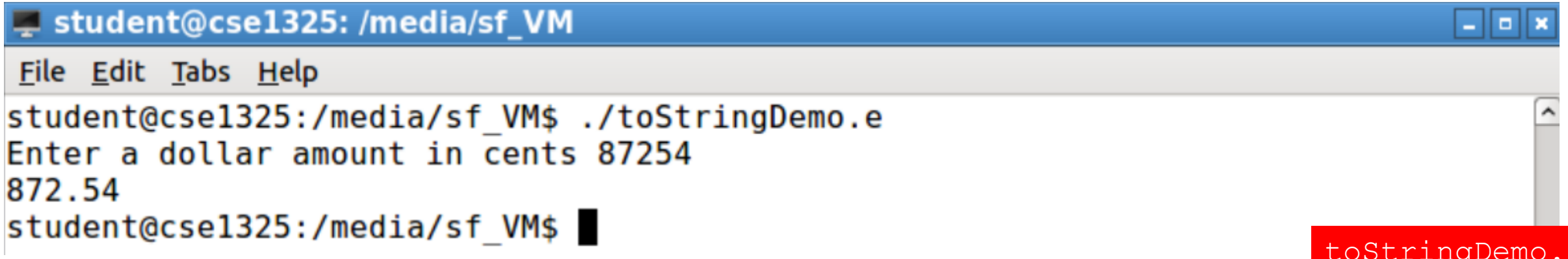
Happy Path is the default scenario where no exceptions or error conditions occur.

Happy Path Testing is only testing with well defined test cases that only uses expected inputs; therefore, achieves only expected results. No exceptions or error conditions occur.

Using only Happy Path Testing will result in your program not being robust and not being user friendly.

to_string()

```
long amount;

cout << "Enter a dollar amount in cents ";
cin >> amount;

std::string dollars{std::to_string(amount / 100)};

std::string cents{std::to_string(std::abs(amount % 100))};

cout << dollars + "." + (cents.size() == 1 ? "0" : "") + cents << endl;
```



toStringDemo.cpp

String Stream Processing

C++ stream I/O includes capabilities for inputting from, and outputting to, `string`s in memory

Class `istringstream`

Supports input from a string

Class `ostringstream`

Supports output to a string

`stringstream` is derived from `iostream` and can be used for both input and output.

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

String Stream Processing

There are two ways to get data into a stringstream…

1. Use the insertion operator <<

```
stringstream os;
os << "This is silly" << endl;
```

2. Use the `str(string)` function to set the value of the buffer

```
stringstream os;
os.str("This is silly");
```

String Stream Processing

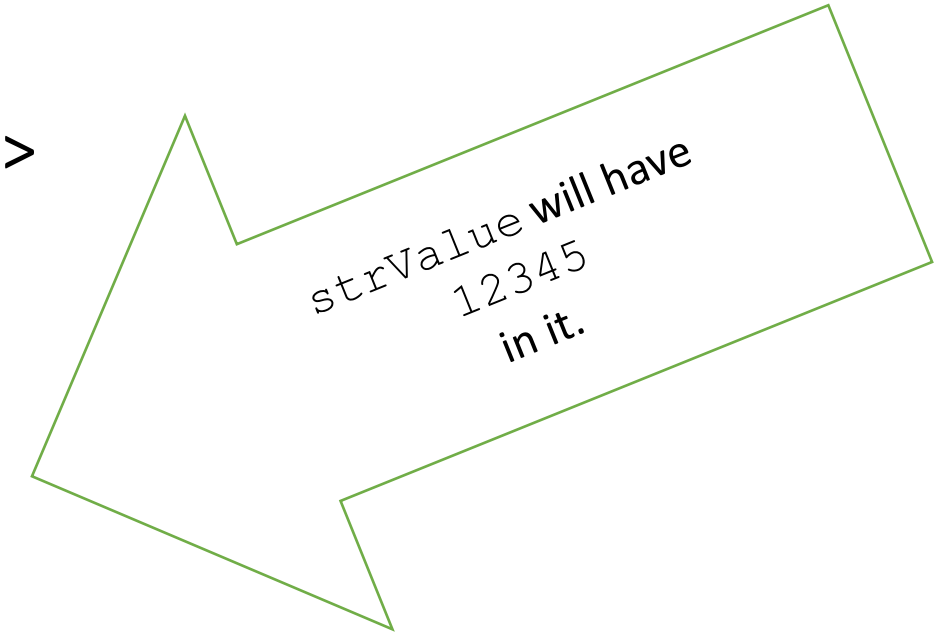
There are two ways to get data out of a stringstream…

1. Use the `str()` function to retrieve the results of the stringstream

```
stringstream os;
os << "12345 67.89" << endl;
cout << os.str();
```

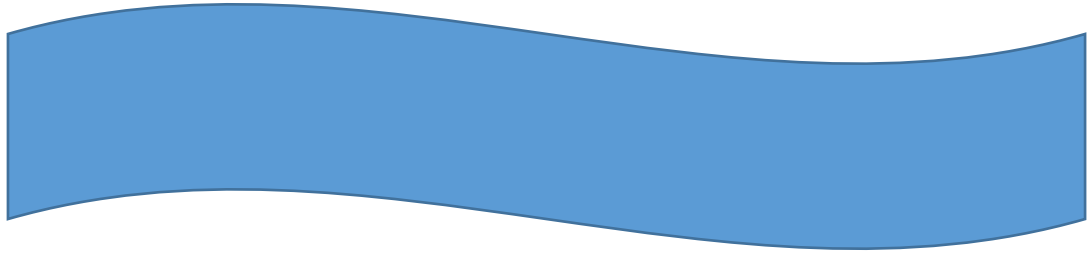
2. Use extraction operator >>

```
stringstream os;
os << "12345 67.89";

string strValue;
os >> strValue;
```



String Stream Processing

```
stringstream os;
os << "12345 67.89";
```



```
string strValue1;
os >> strValue1;
```

extracts all characters from `os` up to the whitespace

```
string strValue2;
os >> strValue2;
```

extracts all characters from `os` after the whitespace

```
cout << strValue1 << " - " << strValue2 << endl;
```

```
12345 - 67.89
```

String Stream Processing

Note that the >> operator iterates through the string…

each successive use of >> returns the next extractable value in the stream.

On the other hand, `str()` returns the whole value of the stream, even if the >> has already been used on the stream.

String Stream Processing

```
string FullGreeting{"Hello_there! How_are_you? I_am_fine. 1 2 3"};
string Greeting1;
string Greeting2;
string Greeting3;
int GNum1, GNum2, GNum3;
stringstream MySS{FullGreeting};

MySS >> Greeting1 >> Greeting2 >> Greeting3
                  >> GNum1 >> GNum2 >> GNum3;

cout << "The " << GNum1 << "st greeting is " << Greeting1 << endl
     << "The " << GNum2 << "nd greeting is " << Greeting2 << endl
     << "The " << GNum3 << "rd greeting is " << Greeting3 << endl;
```

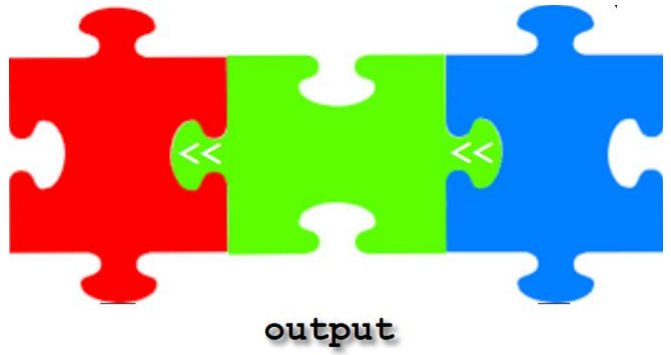
```
The 1st greeting is Hello_there!
The 2nd greeting is How_are_you?
The 3rd greeting is I_am_fine.
```

stringstreamDemo.cpp

String Stream Processing

```
string String1{"Hello there!"};
string String2{"How are you?"};
string String3{"I am fine."};
stringstream SS1;

SS1 << String1 << endl;
cout << SS1.str();

cout << "-------------------------------------" << endl;

SS1 << String2 << endl;
cout << SS1.str();

cout << "-------------------------------------" << endl;

SS1 << String3 << endl;
cout << SS1.str();
```

```
Hello there!
-------------------------------------
Hello there!
How are you?
-------------------------------------
Hello there!
How are you?
I am fine.
```



stringstream2Demo.cpp

String Stream Processing

```
string String1{"Hello there!"};
string String2{"How are you?"};
string String3{"I am fine."};
stringstream SS2;

cout << endl << "Printing in reverse with tabs\n";
SS2 << String3 << "\t" << String2 << "\t" << String1 << endl;
cout << SS2.str();
```

```
Printing in reverse with tabs
I am fine.      How are you?    Hello there!
```

ostringstreamDemo.cpp

String Stream Processing

When reusing a `stringstream` variable, you should set the stream to empty/blanks and call the `clear()` function to clear any flags on the stream.

```
stringstream os;
os << "Hello ";

os.str(""); // erase the buffer
os.clear(); // reset error flags

os << "World!";
cout << os.str();
```