6

CSE 1325

Week of 09/28/2020

Instructor : Donna French

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 `char vector` named `Frog`?

```
vector<char>Frog;
```

a `float vector` named `Toad` with 7 elements?

```
vector<float>Toad(7);
```

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

Note that in both the uninitialized and initialized case, you do not need to include the length at compile time. This is because `std::vector` will dynamically allocate memory for its contents as requested.

When you add to the end of a vector, memory will be dynamically allocated for the new element.

However, if you try to add PAST the end of the vector, the vector will not resize.

vector

```
13              vector<char>Frog;
(gdb)
14              vector<float>Toad(7);
(gdb)
15              vector<bool>Cat{0,1,1,true,0};
(gdb)
17              cout << Toad[0] << endl;
(gdb)
0
18              cout << boolalpha << Cat[0] << endl;
(gdb)
0
19              cout << Frog[0] << endl;
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554fca in main () at vector1Demo.cpp:19
```

How would you print the word “`false`” instead of the 0?

vector1Demo.cpp

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

```
9           vector<char>Frog;
(gdb)
10          vector<float>Toad(7);
(gdb)
11          vector<bool>Cat{0,1,1,true,0};
(gdb)
13          cout << Frog.size() << endl;
(gdb)
0
14          cout << Toad.size() << endl;
(gdb)
7
15          cout << Cat.size() << endl;
(gdb)
5
```

vector

We can copy a vector by creating a new vector and initializing it to the vector we want to copy.

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

```
11          vector<bool>Cat{0,1,1,true,0};
(gdb)
13          vector<bool>Dog{Cat};
```

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

vector

We can copy a vector by using the =.

```
11              vector<bool>Cat{0,1,1,true,0};
(gdb)
12              vector<bool>Rat;
(gdb)
14              vector<bool>Dog{Cat};
(gdb)
16              Rat = Cat;
(gdb) p Cat
$1 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}
(gdb) p Rat
$2 = std::vector<bool> of length 0, capacity 0
(gdb) p Dog
$3 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}
(gdb) p Rat
$4 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}
```

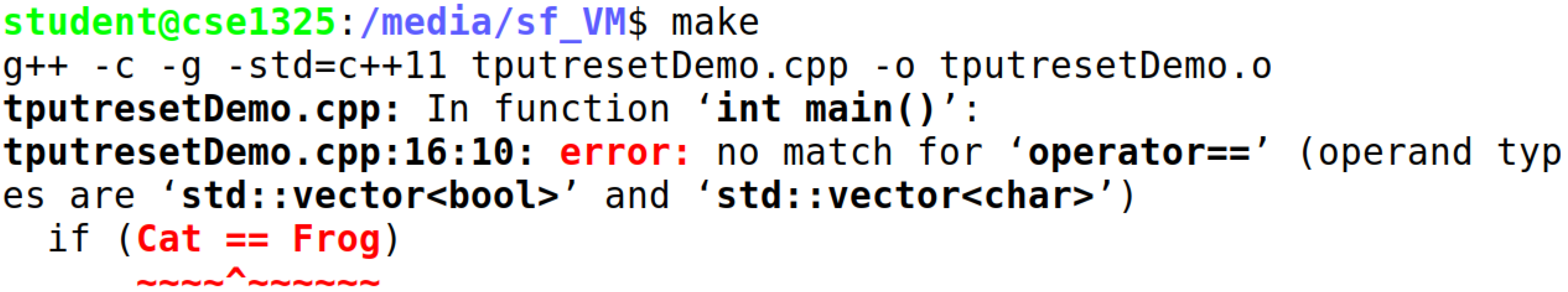
vector

We can compare two vectors of the same type

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

vector<bool>Dog{Cat};

if (Cat == Dog)
        cout << "equal";
else
        cout << "not equal";

Cat[3] = false;

if (Cat == Dog)
        cout << "equal";
else
        cout << "not equal";
```

vector

We cannot compare two vectors of the different types

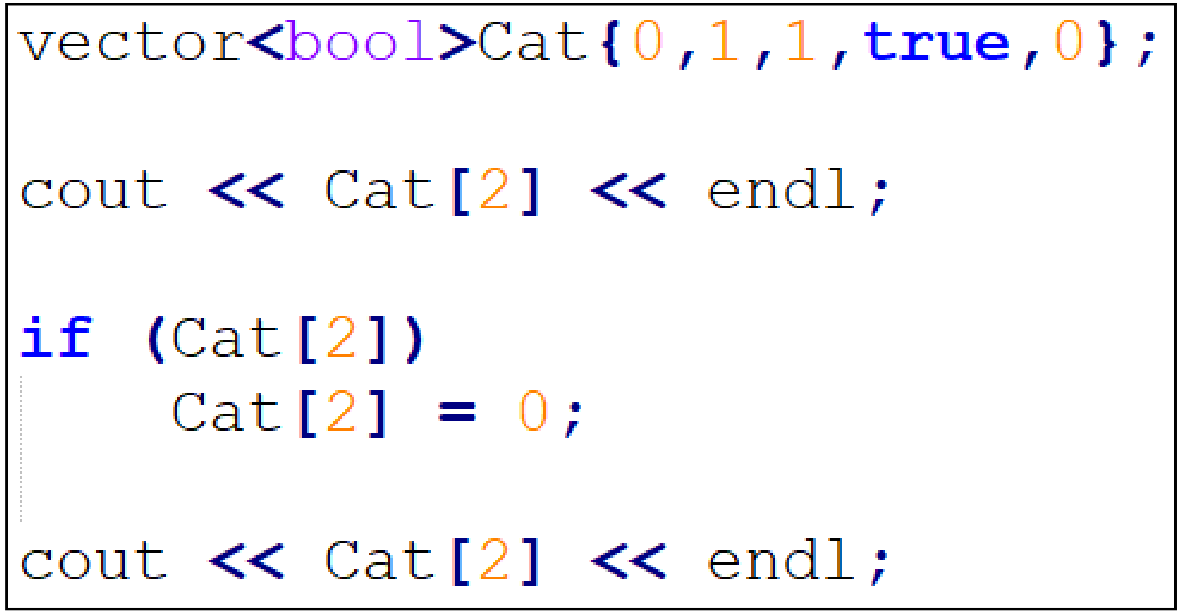


```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 tputresetDemo.cpp -o tputresetDemo.o
tputresetDemo.cpp: In function ‘int main()’:
tputresetDemo.cpp:16:10: error: no match for ‘operator==’ (operand typ
es are ‘std::vector<bool>’ and ‘std::vector<char>’)
  if (Cat == Frog)
      ~~~~^~~~~~~
```

vector

We can use [ ] just like arrays to access and set individual elements.

```
11              vector<bool>Cat{0,1,1,true,0};
(gdb)
13              cout << Cat[2] << endl;
(gdb) p Cat[2]
$1 = true
(gdb) n
1
15              if (Cat[2])
(gdb)
16                      Cat[2] = 0;
(gdb)
18              cout << Cat[2] << endl;
(gdb) p Cat[2]
$2 = false
(gdb) n
0
```



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

cout << Cat[2] << endl;

if (Cat[2])
    Cat[2] = 0;

cout << Cat[2] << endl;
```

vector

Just like in C, the [ ] operator will let us walk over memory.

```
11            vector<bool>Cat{0,1,1,true,0};
(gdb)
(gdb) p Cat
$1 = std::vector<bool> of length 5, capacity 64 = {0, 1, 1, 1, 0}

13            Cat[20] = 1;

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

14            cout << Cat[20] << endl;
1
```

`vector2Demo.cpp`

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)   MyVector.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.

vector

Vector subscripts and `at()` are based on size/length, not capacity

The range for the subscript operator (`[]`) and `at()` function is based on the vector's length, not the capacity.

If a vector has a size/length 3 and a capacity 5, then what happens if we try to access the array element with index 4?

It fails since 4 is greater than the length of the vector.

```
MyVector.push_back(10);

cout << "\n\nMyVector after push_back(10)" << endl;

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

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

```
MyVector after push_back(10)
2    4     6    8     10
MyVector.size() 5 MyVector.capacity() 8
MyVector.push_back(12);

cout << "\n\nMyVector after push_back(12)" << endl;

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

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

```
MyVector after push_back(12)
2     4     6     8     10    12
MyVector.size() 6 MyVector.capacity() 8
```

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

```
cout << "The 1st element  is " << MyVector.front() << endl;
cout << "The last element is " << MyVector.back()  << endl;
cout << "The 3rd element  is " << MyVector.at(3)   << endl;
```

```
The 1st element  is 2
The last element is 12
The 3rd element  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
```

2	4	6	8	10	12

```
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
```

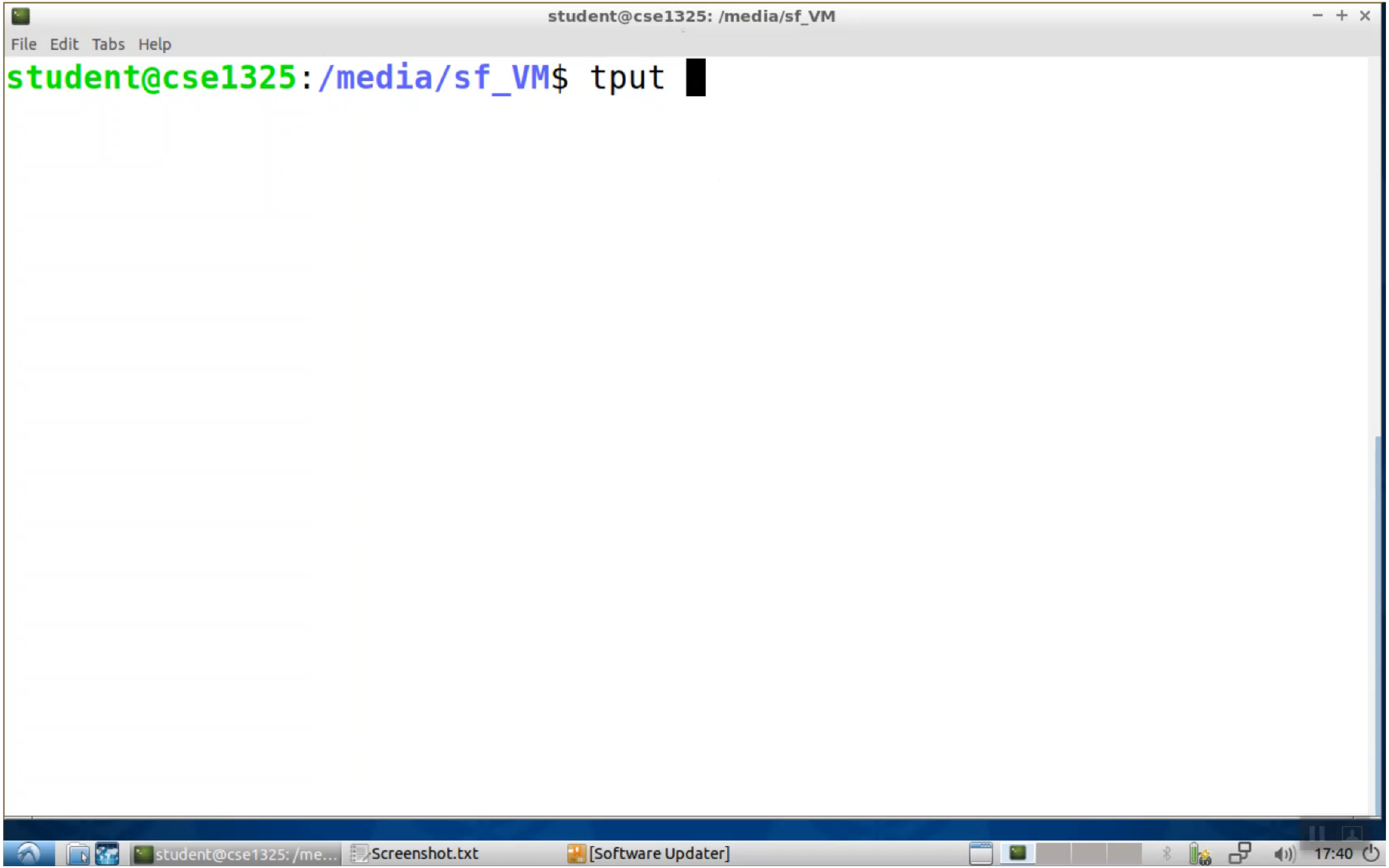
pop_back()

Note that the capacity did not change – did not shrink

```
1.2345 2.3456 3.4567 4.5678 5.6789
size = 5 and capacity = 5
1.2345 2.3456 3.4567 4.5678
size = 4 and capacity = 5
1.2345 2.3456 3.4567
size = 3 and capacity = 5
1.2345 2.3456
size = 2 and capacity = 5
1.2345
size = 1 and capacity = 5

size = 0 and capacity = 5

size = 0 and capacity = 5
```

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

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

Bank.pop_back();
```

How are we popping an empty vector?



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

```
2     4     6     8     10
```

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

Command Line Arguments

Command line arguments are optional string arguments that are passed by the operating system to the program when it is launched.

The program can then use them as input (or ignore them).

Much like function parameters provide a way for a function to provide inputs to another function, command line arguments provide a way for people or programs to provide inputs to a *program*.

Command Line Arguments

Passing command line arguments

Executable programs can be run on the command line by invoking them by name.

```
./Code1_1000074079.e
```

In order to pass command line arguments to a program, we list the command line arguments after the executable name

```
./Code1_1000074079.e FileToRead.txt
```

Command Line Parameters

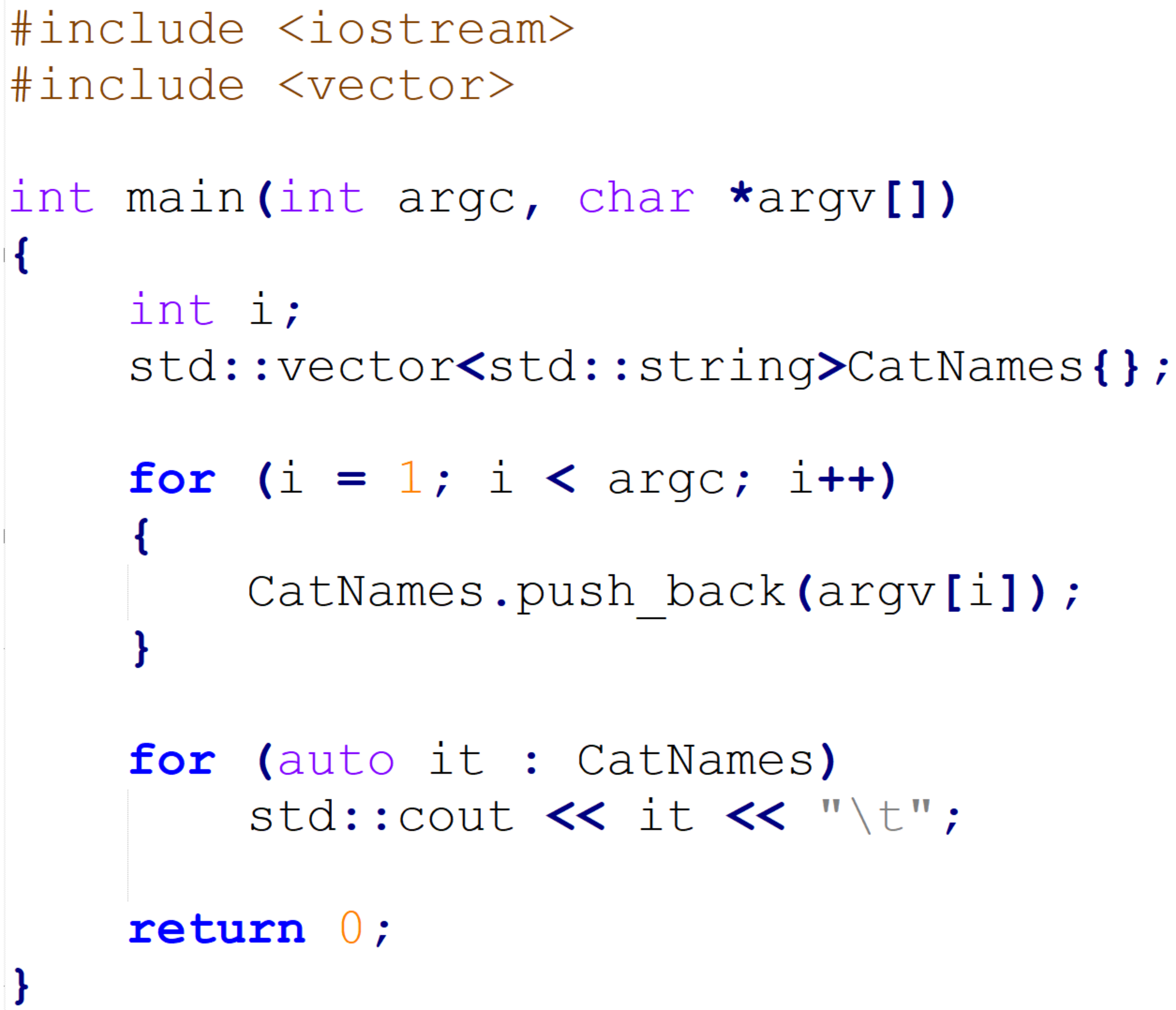
Running a program with command line parameters

```
./Code1_1000074079.e clp1 clp2 clp3
```

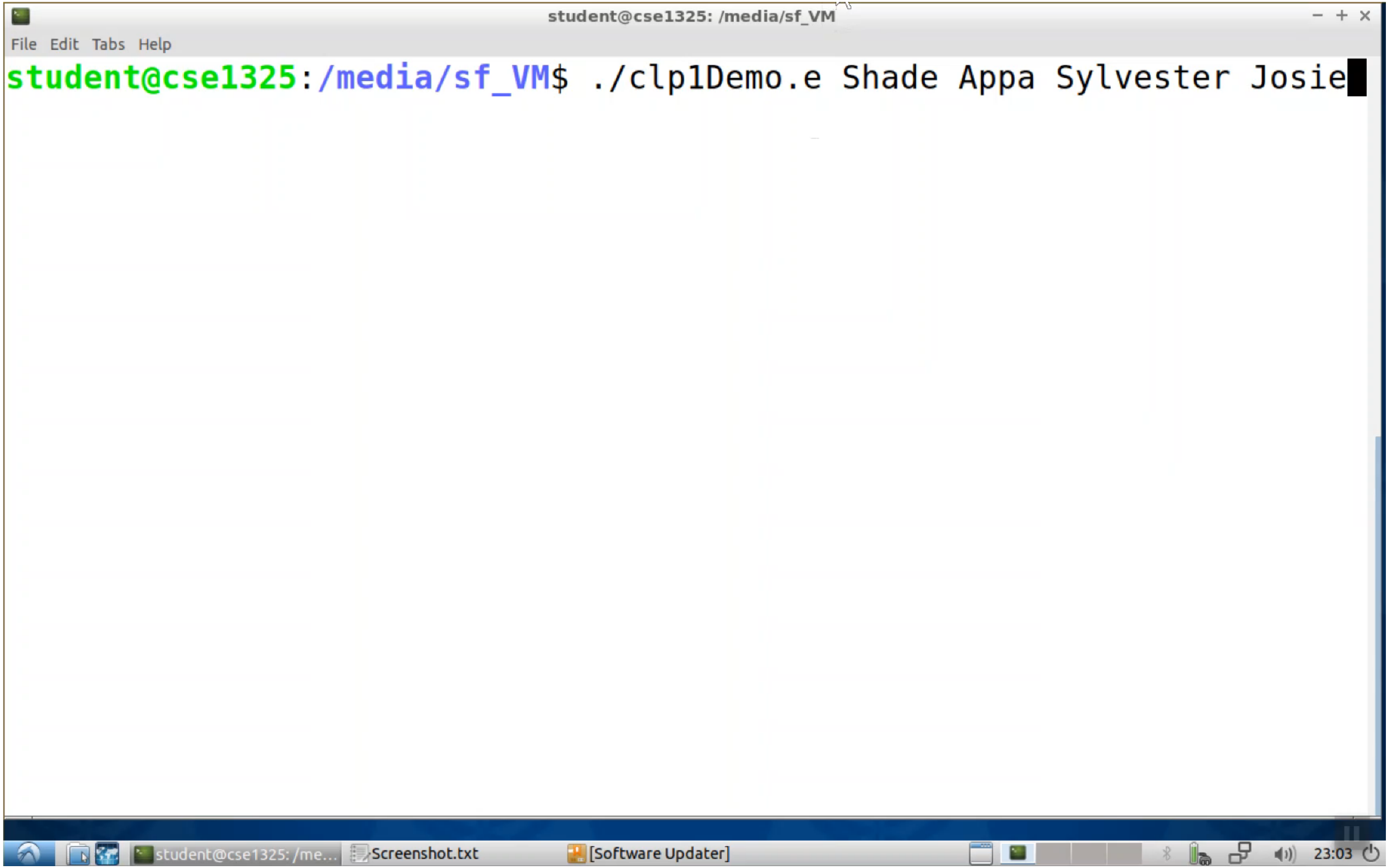
Running a program in debug with command line parameters

```
gdb --args ./Code1_1000074079.e clp1 clp2 clp3
```

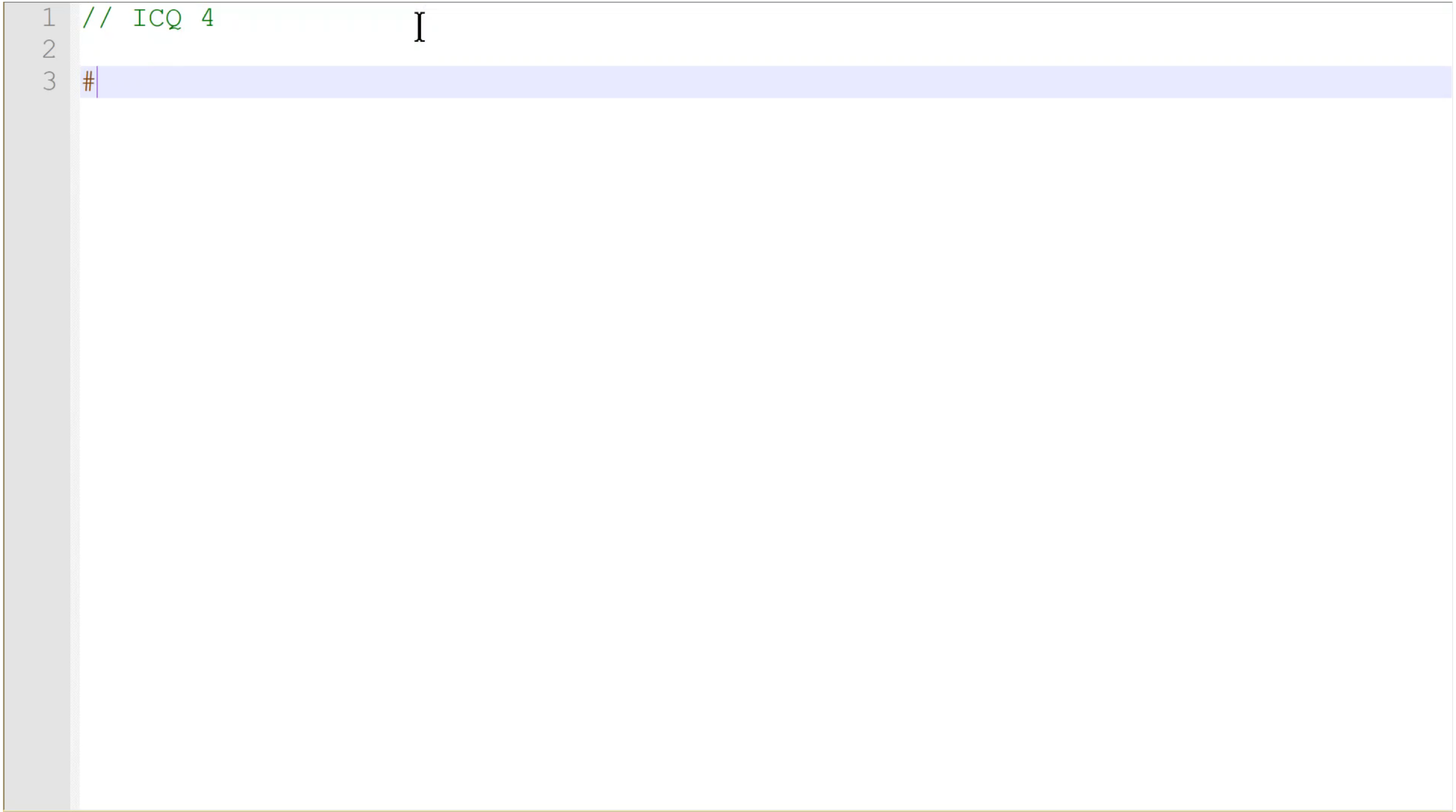
```
student@cse1325:/media/sf_VM$ gdb --args clp1Demo.e FileToRead.txt
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git

Reading symbols from clp1Demo.e...done.
(gdb) break main
Breakpoint 1 at 0xb73: file clp1Demo.cpp, line 7.
(gdb) run
Starting program: /media/sf_VM/clp1Demo.e FileToRead.txt

Breakpoint 1, main (argc=2, argv=0x7fffffffe138) at clp1Demo.cpp:7
7	{
(gdb) p argc
$1 = 2
(gdb) p *argv
$2 = 0x7fffffffe440 "/media/sf_VM/clp1Demo.e"
(gdb) p *argv@argc
$3 = {0x7fffffffe440 "/media/sf_VM/clp1Demo.e",
  0x7fffffffe458 "FileToRead.txt"}
(gdb) p argv[0]
$4 = 0x7fffffffe440 "/media/sf_VM/clp1Demo.e"
(gdb) p argv[1]
$5 = 0x7fffffffe458 "FileToRead.txt"
```



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

int main(int argc, char *argv[])
{
    int i;
    std::vector<std::string>CatNames{};

    for (i = 1; i < argc; i++)
    {
        CatNames.push_back(argv[i]);
    }

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

    return 0;
}
```



```
student@cse1325:/media/sf_VM$ ./clp1Demo.e Shade Appa Sylvester Josie
```



```
// ICQ 4

#
```

Default Arguments

A **default argument** is a default value provided for a function parameter.

If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used.

If the user does supply an argument for the parameter, the user-supplied argument is used.

Default Arguments

If a function is repeatedly invoked with the same argument value for a particular parameter, then

you can specify that such a parameter has a default argument

Default argument – a default value is passed to that parameter

When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument.

```
unsigned int boxVolume(unsigned int length=1, unsigned int width=1, unsigned int height=1)
{
	return length * width * height;
}

int main()
{
	// nothing is passed - use defaults for all
	cout << "boxVolume() = " << boxVolume() << endl;

	// length is passed - use default width and height
	cout << "\n\nboxVolume(10) = " << boxVolume(10) << endl;

	// length and width are passed - use default height
	cout << "\n\nboxVolume(10,5) = " << boxVolume(10,5) << endl;

	// length and width and height are all passed - no defaults
	cout << "\n\nboxVolume(10,5,2) = " << boxVolume(10,5,2) << endl;

	return 0;
}
```

default values

Default Arguments

Any arguments passed to the function explicitly are assigned to the function's parameters from left to right.

`boxVolume`'s parameters (in order) are `length`, `width`, `height`

So when `boxVolume()` receives one argument, it assigns the value of that argument to its leftmost parameter which is `length`.

`boxVolume(10)`

10 is assigned to `length`

`boxVolume(10,5)`

10 is assigned to `length` and 5 is assigned to `width`

Default Arguments

Default values need to be specified in EITHER the prototype or the function **BUT** not both.

```
unsigned int boxVolume(unsigned int length, unsigned int width, unsigned int height);

unsigned int boxVolume(unsigned int length=1, unsigned int width=1, unsigned int height=1)
{
        return length * width * height;
}
```

OR

Preferred method – just makes the defaults easier to find in the program

```
unsigned int boxVolume(unsigned int length = 1, unsigned int width = 1, unsigned int height = 1);

unsigned int boxVolume(unsigned int length, unsigned int width, unsigned int height)
{
        return length * width * height;
}
```

Default Arguments

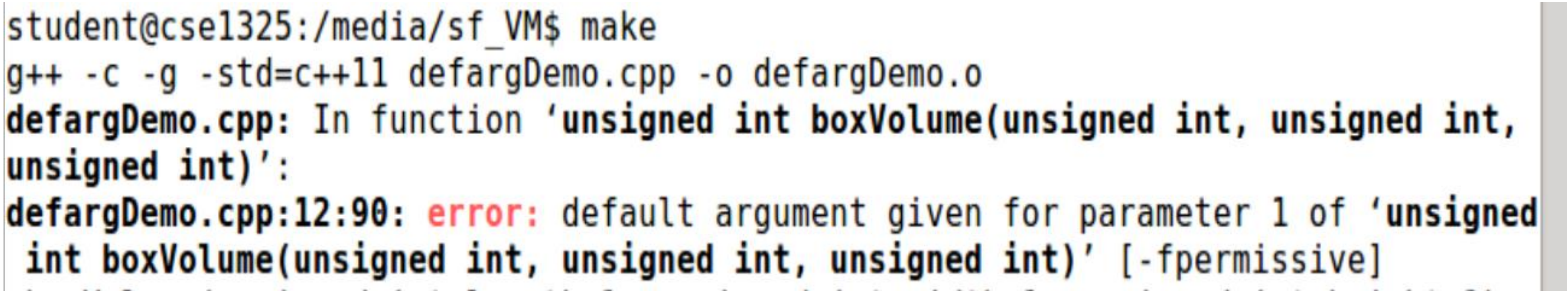
```
unsigned int boxVolume(unsigned int length=1,
unsigned int width=1, unsigned int height=1);
```

```
unsigned int boxVolume(unsigned int length=1,
unsigned int width=1, unsigned int height=1)
{
    return length * width * height;
}
```

Default Arguments

All default arguments must be for the rightmost parameters.

```
// Want to use default length but pass in width and height
cout << "\n\nboxVolume(,5,2) = " << boxVolume(,5,2) << endl;
```



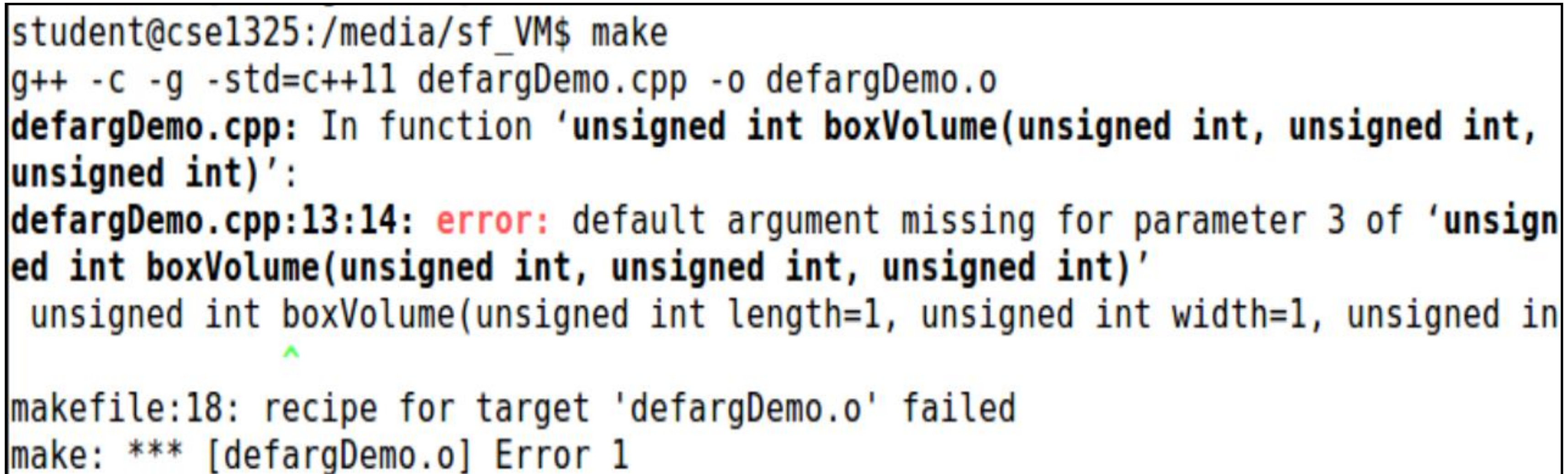
```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 defargDemo.cpp -o defargDemo.o
defargDemo.cpp: In function ‘unsigned int boxVolume(unsigned int, unsigned int,
unsigned int)’:
defargDemo.cpp:12:90: error: default argument given for parameter 1 of ‘unsigned
 int boxVolume(unsigned int, unsigned int, unsigned int)’ [-fpermissive]
```

Default Arguments

Default arguments must be the rightmost (trailing) arguments in a function's parameter list.

no default value

```
unsigned int boxVolume(unsigned int length=1, unsigned int width=1,unsigned int height)
{
   return length * width * height;
}
```



```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 defargDemo.cpp -o defargDemo.o
defargDemo.cpp: In function ‘unsigned int boxVolume(unsigned int, unsigned int,
unsigned int)’:
defargDemo.cpp:13:14: error: default argument missing for parameter 3 of ‘unsign
ed int boxVolume(unsigned int, unsigned int, unsigned int)’
 unsigned int boxVolume(unsigned int length=1, unsigned int width=1, unsigned in
              ^
makefile:18: recipe for target 'defargDemo.o' failed
make: *** [defargDemo.o] Error 1
```

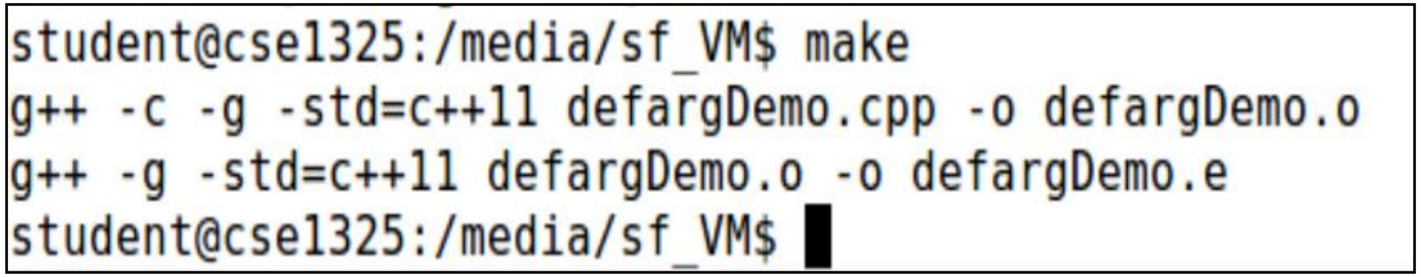
Default Arguments

no default value

Default arguments must be the rightmost (trailing) arguments in a function's parameter list.

```
unsigned int boxVolume(unsigned int length, unsigned int width=1,unsigned int height=1)
{
   return length * width * height;
}
```

OK because `length` is the leftmost.



```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 defargDemo.cpp -o defargDemo.o
g++ -g -std=c++11 defargDemo.o -o defargDemo.e
student@cse1325:/media/sf_VM$ █
```

Default Arguments

Default values can be any expression, including constants, global variables or function calls.

```
#define FROG 2
unsigned int boxVolume(unsigned int length=1, unsigned
int width=FROG, unsigned int height=1)
```

```
vector<int>TOAD(4);
unsigned int boxVolume(unsigned int length=1, unsigned
int width=TOAD.size(), unsigned int height=1)
```

Default Arguments

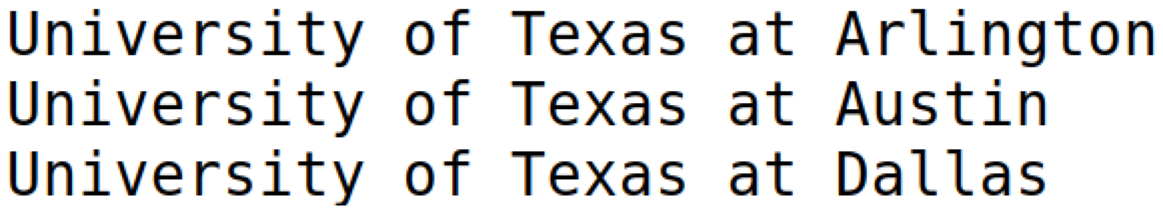
```
#include <iostream>

using namespace std;

void PrintIT(string Word1="University of Texas", string Word2=" at ", string
Word3);

void PrintIT(string Word1, string Word2, string Word3)
{
      cout << Word1 << Word2 << Word3;
}


int main()
{
      PrintIT("University of Texas", " at ","Arlington");
      PrintIT("University of Texas", " at ","Austin");
      PrintIT("University of Texas", " at ","Dallas");

      return 0;
}
```

ile?

NO

Default Arguments

```
#include <iostream>

using namespace std;

void PrintIT(string Word3, string Word1="University of Texas", string
Word2=" at ");

void PrintIT(string Word3, string Word1, string Word2)
{
   cout << Word1 << Word2 << Word3 << endl;
}


int main()
{
   PrintIT("Arlington");
   PrintIT("Austin");
   PrintIT("Dallas");

   return 0;
}
```

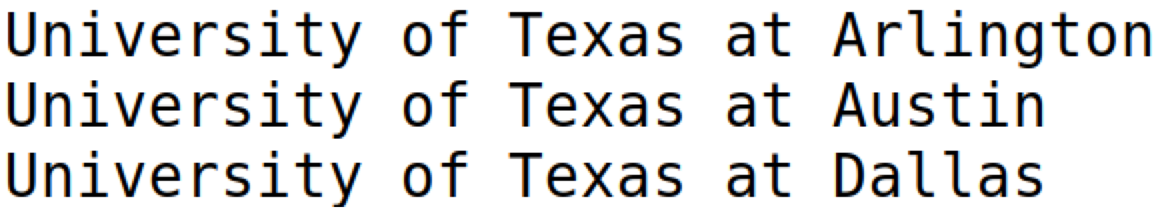


```
University of Texas at Arlington
University of Texas at Austin
University of Texas at Dallas
```

Will this compile?

Yes

Default Arguments



```
University of Texas at Arlington
University of Texas at Austin
University of Texas at Dallas
```

```
#include <iostream>

using namespace std;

void PrintIT(string Word3, string Word1="University of Texas", string Word2="
at ");

void PrintIT(string Word3, string Word1, string Word2)
{
   cout << Word1 << Word2 << Word3 << endl;
}


int main()
{
   PrintIT("Arlington");
   PrintIT("Austin");
   PrintIT("Dallas");

   return 0;
}
```

How to print

```
University of Tulsa
PrintIT("", "University of Tulsa", "");
Texas A&M University-Commerce
PrintIT("Commerce", "Texas A&M University", "-");
Texas A&M University at Galveston
PrintIT("Galveston", "Texas A&M University");
```

Function Overloading

Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters.

Consider this function…

```
int funA(int X, int Y, int Z)
{
 return X+Y+Z;
}
```

Function Overloading

```
int funA(int X, int Y, int Z)
{
        return X+Y+Z;
}

int main(void)
{
        cout << funA(2,2,2) << endl;

        cout << funA(3.3,3.3,3.3) << endl;

        return 0;
}
```

This would print “6”

This would print “9”. Why?

Function Overloading

What if we created a function with the same name but used a different type for the parameters?

```
int funA(int X, int Y, int Z)
{
    return X+Y+Z;
}
```

Prints “6”

```
double funA(double X, double Y, double Z)
{
    return X+Y+Z;
}
```

Prints “9.9”

Function Overloading

Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters.

The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.

The combination of a function's name and its parameters types and the order of them is called a **signature**.

Function Overloading

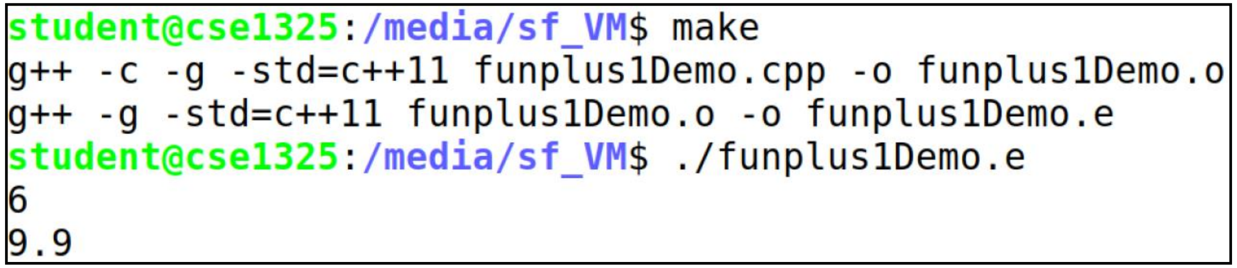
```
int funA(int X, int Y, int Z)
{
  return X+Y+Z;
}
```

signature – funA+int+int+int

```
double funA(double X, double Y, double Z)
{
  return X+Y+Z;
}
```

signature – funA+double+double+double

```
int main(void)
{
  cout << funA(2,2,2) << endl;
  cout << funA(3.3,3.3,3.3) << endl;

  return 0;
}
```



```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 funplus1Demo.cpp -o funplus1Demo.o
g++ -g -std=c++11 funplus1Demo.o -o funplus1Demo.e
student@cse1325:/media/sf_VM$ ./funplus1Demo.e
6
9.9
```

Function Overloading

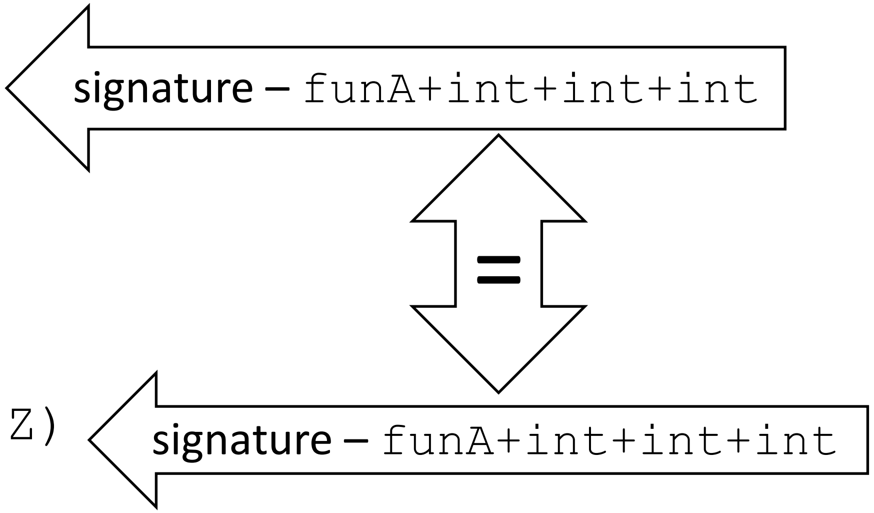
What is the signature of each of these functions?

```
std::string displayMoney(int amount)
     signature = displayMoney+int


bool buyPencils(int payment, std::string& change, int& action,
const int &quantity, int &inventoryLevel, int &changeLevel)
     signature=buyPencils+int+string+int+const int+int+int


int PencilMenu()
     signature = PencilMenu
```

Function Overloading

Creating overloaded functions with identical parameter lists and different return types is a compilation error.

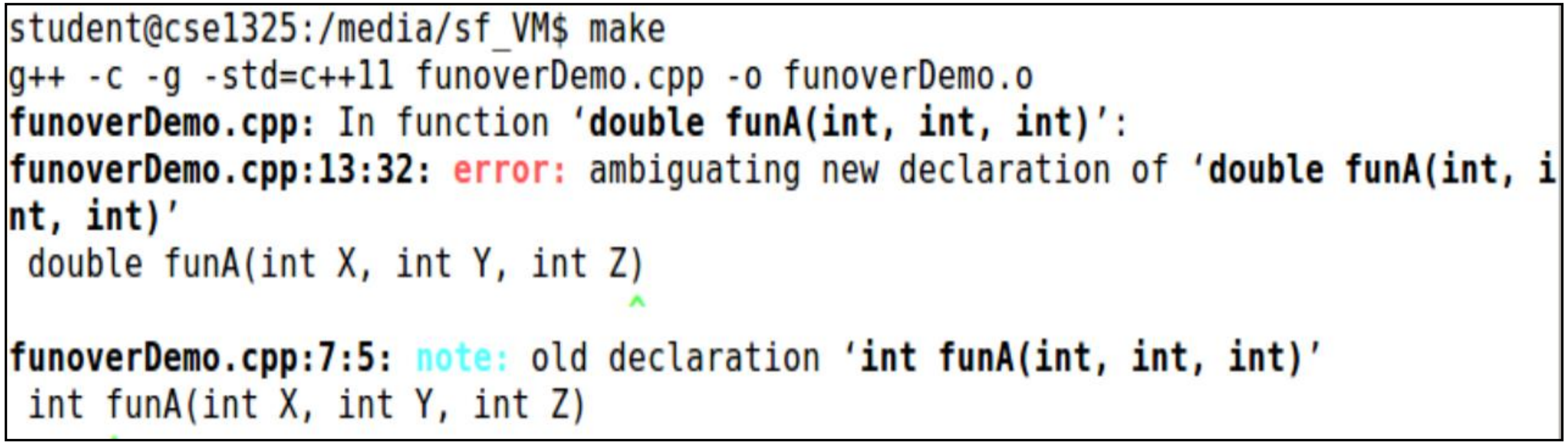
```
int funA(int X, int Y, int Z)
{
  return X*Y*Z;
}
double funA(int X, int Y, int Z)
{
  return X*Y*Z;
}
```



signature – funA+int+int+int

=

signature – funA+int+int+int



```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 funoverDemo.cpp -o funoverDemo.o
funoverDemo.cpp: In function ‘double funA(int, int, int)’:
funoverDemo.cpp:13:32: error: ambiguating new declaration of ‘double funA(int, i
nt, int)’
 double funA(int X, int Y, int Z)
                                ^

funoverDemo.cpp:7:5: note: old declaration ‘int funA(int, int, int)’
 int funA(int X, int Y, int Z)
```

Function Overloading

Function overloading can lower a program’s complexity significantly while introducing very little additional risk.

Function overloading typically works transparently and without any issues.

The compiler will flag all ambiguous cases, and they can generally be easily resolved.

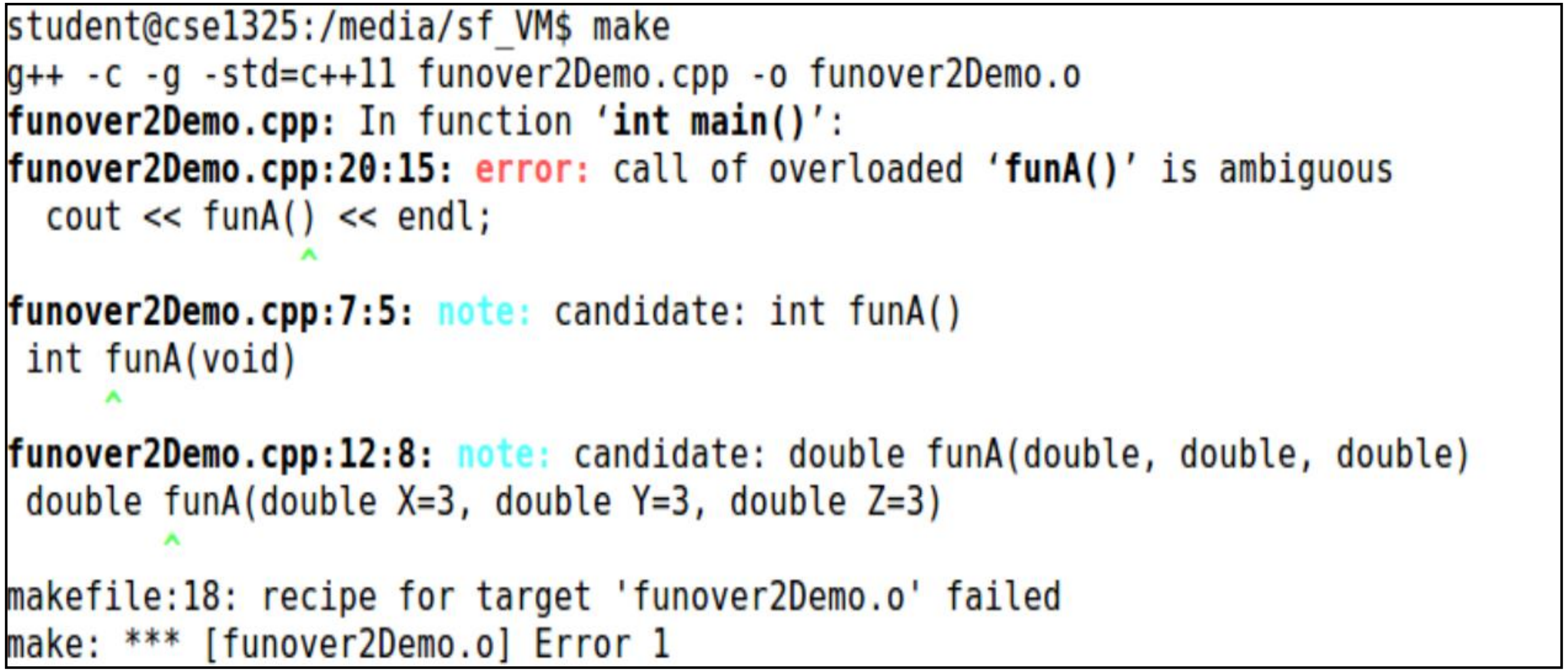
Conclusion : function overloading can make your program simpler.

Function Overloading and Default Arguments

A function with default arguments omitted might be called identically to another overloaded function.

```
int funA(void)
{
        return 3;
}
```

```
cout << funA() << endl;
cout << funA() << endl;
```



```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 funover2Demo.cpp -o funover2Demo.o
funover2Demo.cpp: In function ‘int main()’:
funover2Demo.cpp:20:15: error: call of overloaded ‘funA()’ is ambiguous
  cout << funA() << endl;
               ^
funover2Demo.cpp:7:5: note: candidate: int funA()
 int funA(void)
     ^
funover2Demo.cpp:12:8: note: candidate: double funA(double, double, double)
 double funA(double X=3, double Y=3, double Z=3)
        ^
makefile:18: recipe for target 'funover2Demo.o' failed
make: *** [funover2Demo.o] Error 1
```

```
double funA(double X=3, double Y=3, double Z=3)
{
        return X*Y*Z;
}
```

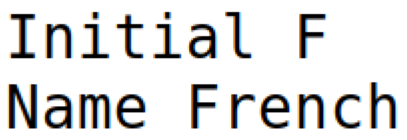
```
cout << funA(3) << endl;
cout << funA(3) << endl;
27
27
```

Function Overloading and Default Arguments

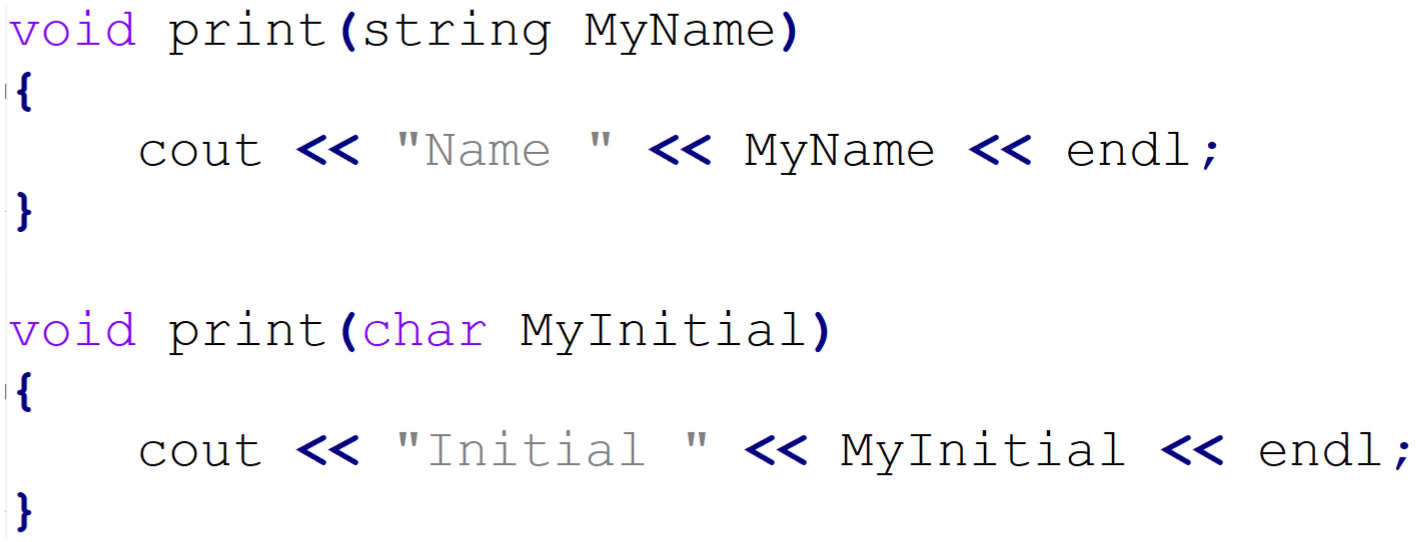
Functions with default arguments can be overloaded.

```
void print(string MyName);
void print(char MyInitial='F');
```

```
print();
print("French");
```



```
Initial F
Name French
```

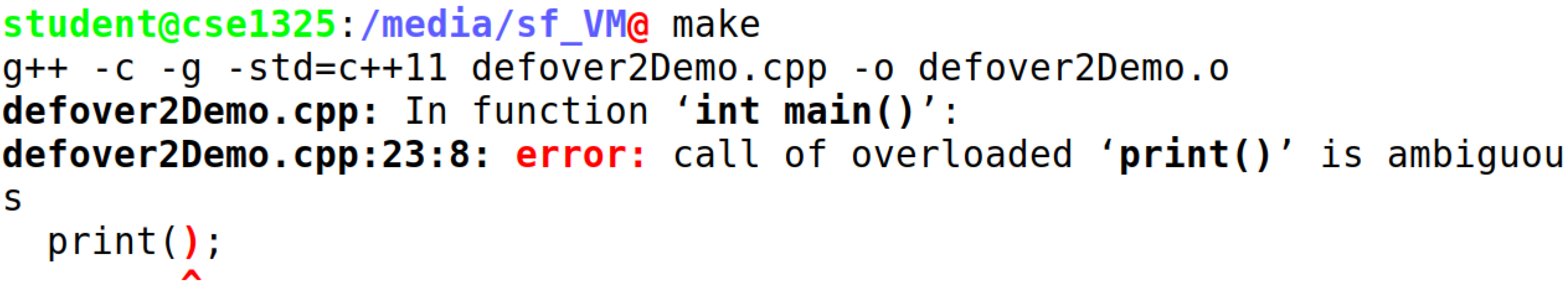


```
void print(string MyName)
{
    cout << "Name " << MyName << endl;
}

void print(char MyInitial)
{
    cout << "Initial " << MyInitial << endl;
}
```

Function Overloading and Default Arguments

It is important to note that default arguments do NOT count towards the parameters that make the function's signature.

```
void print(char Initial1='D', char Initial2='M', char Initial3='F');
void print(char MyInitial='F');
```



```
student@cse1325:/media/sf_VM@ make
g++ -c -g -std=c++11 defover2Demo.cpp -o defover2Demo.o
defover2Demo.cpp: In function ‘int main()’:
defover2Demo.cpp:23:8: error: call of overloaded ‘print()’ is ambiguou
s
  print();
        ^
```

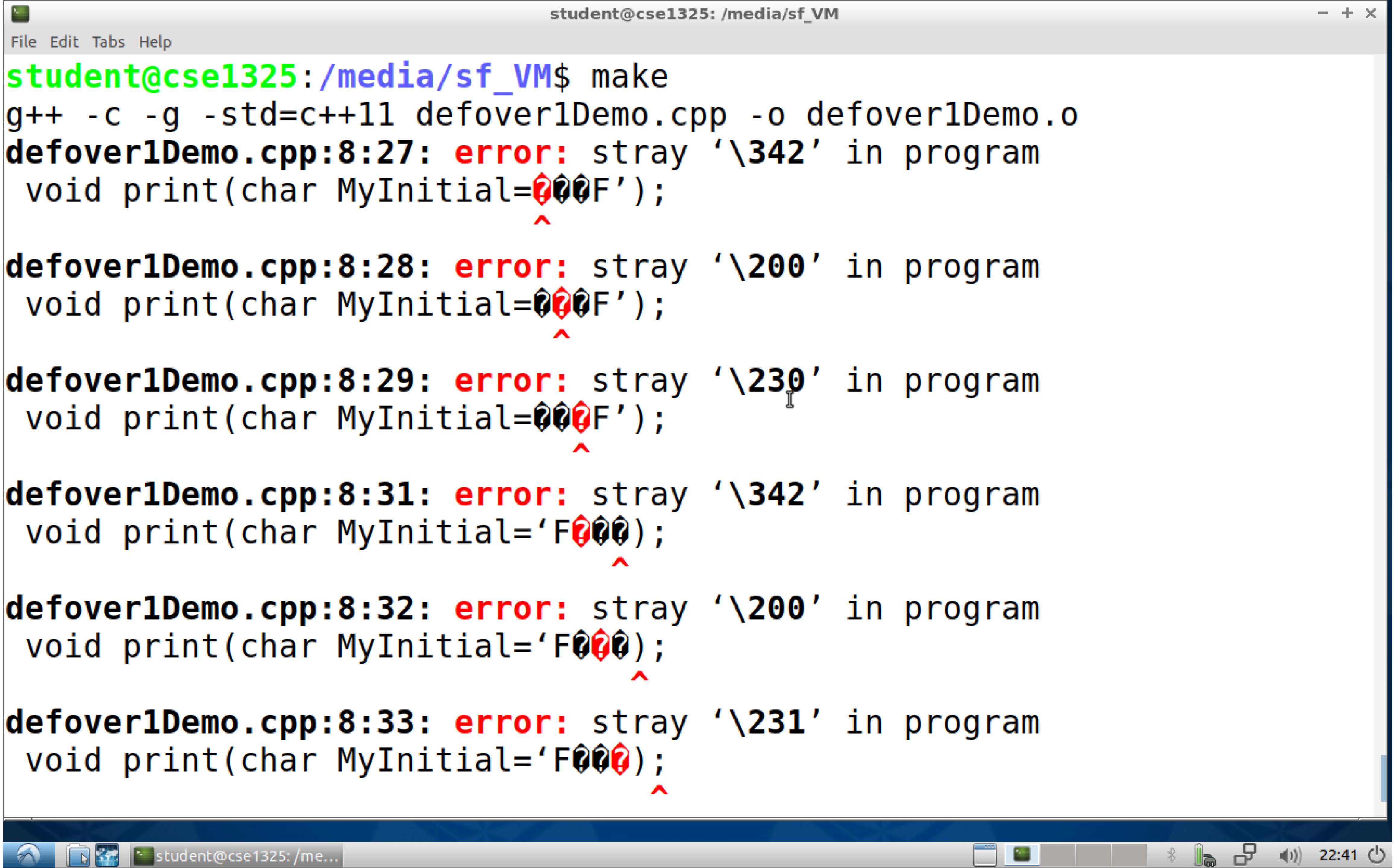
Function Overloading and Default Arguments

```
void print(char Initial1, char Initial2='M', char Initial3='F');
void print(char MyInitial='F'); ☹
```

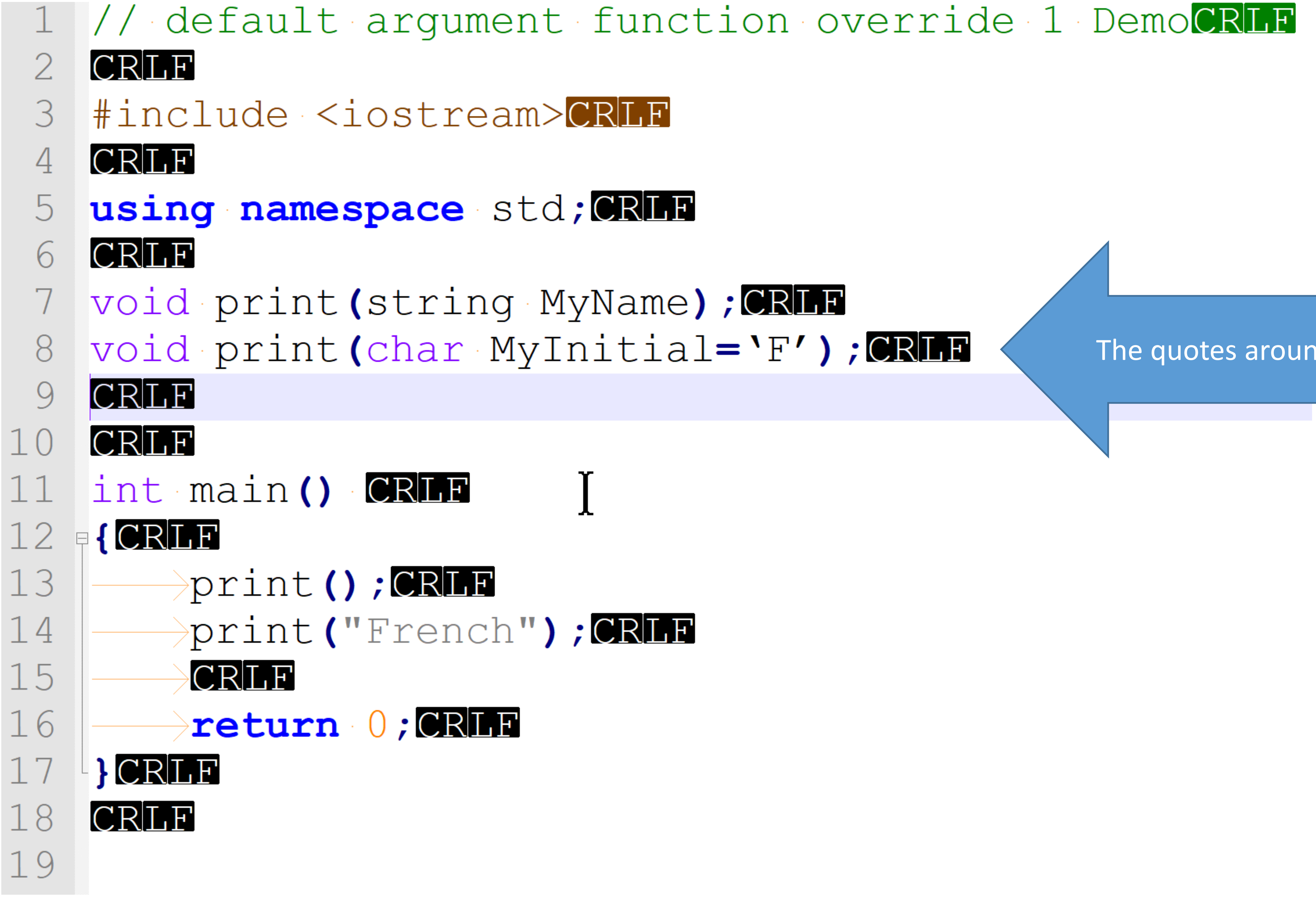
```
void print(char Initial1, char Initial2='M', char Initial3='F');
void print(char MyInitial); ☹
```

```
void print(char Initial1, char Initial2, char Initial3='F');
void print(char MyInitial='F'); ☺
```

```
print('F');
```



```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 defover1Demo.cpp -o defover1Demo.o
defover1Demo.cpp:8:27: error: stray ‘\342’ in program
 void print(char MyInitial=���F’);
                           ^

defover1Demo.cpp:8:28: error: stray ‘\200’ in program
 void print(char MyInitial=���F’);
                            ^

defover1Demo.cpp:8:29: error: stray ‘\230’ in program
 void print(char MyInitial=���F’);
                             ^

defover1Demo.cpp:8:31: error: stray ‘\342’ in program
 void print(char MyInitial=‘F���);
                               ^

defover1Demo.cpp:8:32: error: stray ‘\200’ in program
 void print(char MyInitial=‘F���);
                                ^

defover1Demo.cpp:8:33: error: stray ‘\231’ in program
 void print(char MyInitial=‘F���);
                                 ^
```



```cpp
// default argument function override 1 Demo

#include <iostream>

using namespace std;

void print(string MyName);
void print(char MyInitial=‘F’);


int main() 
{
    print();
    print("French");
    
    return 0;
}

```

The quotes around the F

Function Templates

Overloaded functions are normally used to perform *similar* operations that involve *different* program logic on different data types.

If the program logic and operations are *identical* for each data type, overloading may be performed more compactly and conveniently by using **function templates**.

You write a single function template definition.

Given the argument types provided in calls to your function, C++ automatically generates separate **function template specializations** to handle each type of call appropriately.

```cpp
int int1, int2, int3;
double double1, double2, double3;
char char1, char2, char3;

// call maximum with int
cout << "Input three integer values: ";
cin >> int1 >> int2 >> int3;
cout << "The max integer value is: " << maximum(int1, int2, int3);

// call maximum with double
cout << "\n\nInput three double values: ";
cin >> double1 >> double2 >> double3;
cout << "The max double value is: " << maximum(double1, double2, double3);

// call maximum with char
cout << "\n\nInput three characters: ";
cin >> char1 >> char2 >> char3;
cout << "The max char value is: " << maximum(char1, char2, char3) << endl;
```

funtempDemo.cpp

keyword

```
template <typename   T  >
T    maximum(  T   value1,    T   value2,    T   value3)
{
     T      maximumValue{value1}; // assume value1 is maximum

     if (value2 > maximumValue)
     {
        maximumValue = value2;
     }

     if (value3 > maximumValue)
     {
        maximumValue = value3;
     }

     return maximumValue;
}
```

funtempDemo.h

`<typename T>`
is the template parameter which represents a type that has not yet been specified

Function name `maximum` returns type `T`

`value1`, `value2` and `value3` are of type `T`

`maximumValue` is of type `T`

Function Templates

The compiler creates a separate function definition for each

one expecting three `int` values

function template specialization for type `int`

one expecting three `double` values

function template specialization for type `double`

one expecting three `char` values

function template specialization for type `char`

Function template specialization for type `int`

```
int maximum(int value1, int value2, int value3)
{
   int maximumValue{value1}; // assume value1 is maximum

   if (value2 > maximumValue)
   {
      maximumValue = value2;
   }

   if (value3 > maximumValue)
   {
      maximumValue = value3;
   }

   return maximumValue;
}
```