CSE 1325

Week of 09/07/2020

Instructor : Donna French

Uniform Initialization

Unsafe Conversions

C++ allows for (implicit) unsafe conversions.

unsafe = a value can be implicitly turned into a value of another type that does not equal the original value

```
int IntVar1 = 32112;
char CharVarA = IntVar1;
int IntVar2 = CharVarA;
```

Uniform Initialization

Unsafe Conversions

To be warned against these unsafe conversions, use the uniform initialization format

```
int IntVar1 {32112};
char CharVarA {IntVar1};
int IntVar2 {CharVarA};
```

Terminal - student@maverick: /media/sf_VM

File Edit View Terminal Tabs Help

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

Uniform Initialization

Additional Notes about Uniform Initialization

Type bool can be initialized with UI

```
bool b1 {true};
bool b2 {false};
bool b3 {!true};
bool b4 {!false};
```

A function can be called that returns a value inside the {}

Use empty braces {} to initialize a variable to 0.

student@cse1325: /media/sf_VM

File Edit Tabs Help

```
student@cse1325:/media/sf_VM$ more uui3Demo.cpp
#include <iostream>

using namespace std;

int getValueFromUser()
{
        cout << "Enter an integer: ";
        int input{};
        cin >> input;

        return input;
}

int main()
{
        int num {getValueFromUser()};

        cout << num << " doubled is: " << num * 2 << '\n';

        return 0;
}
student@cse1325:/media/sf_VM$ 
```

DRY vs WET Coding

DRY

Don't Repeat Yourself

Advantages

- Maintainability
- Readability
- Reuse
- Cost
- Testing

WET

Write Everything Twice

We Enjoy Typing

Advantages

NONE

Abstraction

In order to use a function, you only need to know its name, inputs, outputs, and where it lives.

You don't need to know how it works, or what other code it's dependent upon to use it.

This lowers the amount of knowledge required to use other people's code (including everything in the standard library).

Required Formatting of Code

The opening brace for a function should be given its own line and the closing brace should line up with the opening brace. Any code lines within the braces should be indented the same amount which must be between 3 and 5 spaces.

```
int main()
{
    my first line
    my second line
    my third line
}
```

`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

Declaring and initializing a string in one line

```
string MyString("Silly");
```

constructing

A string can also be declared and then assigned a value

```
string MyString;
MyString = "Silly";
```

assignment

`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

New keywords in C++

`const`

`const` variables must be initialized when you define them and then that value can not be changed via assignment.

`const` variables can be initialized from other variables (including non-const ones).

We will use `const` with function parameters when we learn about passing by value in C++.

```
#include <iostream>

using namespace std;

int main()
{
  int x;

  x = 1;

  return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
  const int x;

  x = 1;

  return 0;
}
```

```
constDemo.cpp: In function 'int main()':
constDemo.cpp:7:12: error: uninitialized const 'x' [-fpermissive]
  const int x;
            ^
constDemo.cpp:9:4: error: assignment of read-only variable 'x'
  x = 1;
    ^
makefile:15: recipe for target 'constDemo.o' failed
make: *** [constDemo.o] Error 1
```

```cpp
#include <iostream>

using namespace std;

int main()
{
  const int x = 1;

  x = 1;

  return 0;
}
```

```
constDemo.cpp: In function 'int main()':
constDemo.cpp:9:4: error: assignment of read-only variable 'x'
  x = 1;
    ^
makefile:15: recipe for target 'constDemo.o' failed
make: *** [constDemo.o] Error 1
```

Passing Parameters to Functions

Two basic methods of passing parameters to functions

- *pass by value*
  - parameter is called *value parameter*
  - a copy is made of the current value of the parameter
  - operations in the function are done on the copy – the original does not change

- *pass by reference*
  - parameter is called a *variable parameter*
  - the address of the parameter's storage location is known in the function
  - operations in the function are done directly on the parameter

Passing Parameters to Functions

In C

all parameters are passed by value

the ability to pass by reference does not exist

Pass by reference can be simulated

- pass the address of the variable
- address cannot be modified
- contents of address can be modified

Review from 1320

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

  printf("Before PassByValue call\tMyMainNum = %d\n", MyMainNum);
  PassByValue(MyMainNum);
  printf("After  PassByValue call\tMyMainNum = %d\n", MyMainNum);

  printf("Before PassByRef   call\tMyMainNum = %d\n", MyMainNum);
  PassByRef(&MyMainNum);
  printf("After  PassByRef   call\tMyMainNum = %d\n", MyMainNum);

  return 0;
}
```

Review from 1320

A copy is passed

```
int PassByValue(int MyNum)
{
	MyNum += 100;
	printf("Inside PassByValue\tMyNum      = %d\n", MyNum);
}
```

The address of the actual variable is passed

```
int PassByRef(int *MyNumPtr)
{
	*MyNumPtr += 100;
	printf("Inside PassByRef\tMyRefNum  = %d\n", *MyNumPtr);
}
```

Review from 1320

```
int MyMainNum = 0;

printf("Before PassByValue call\tMyMainNum = %d\n", MyMainNum);
PassByValue(MyMainNum);
printf("After  PassByValue call\tMyMainNum = %d\n", MyMainNum);

int PassByValue(int MyNum)
{
     MyNum += 100;
     printf("Inside PassByValue\tMyNum      = %d\n", MyNum);
}
```

```
Before PassByValue call MyMainNum = 0
Inside PassByValue      MyNum     = 100
After  PassByValue call MyMainNum = 0
```

Review from 1320

```
int MyMainNum = 0;

printf("Before PassByRef   call\tMyMainNum = %d\n", MyMainNum);
PassByRef(&MyMainNum);
printf("After  PassByRef   call\tMyMainNum = %d\n", MyMainNum);

int PassByRef(int *MyNumPtr)
{
     *MyNumPtr += 100;
     printf("Inside PassByRef\tMyNumPtr  = %d\n", *MyNumPtr);
}

Before PassByRef   call MyMainNum = 0
Inside PassByRef        MyRefNum  = 100
After  PassByRef   call MyMainNum = 100
```

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;
}
```

What happens if we remove the &?

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

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.

namespace

What is a namespace?

A namespace defines an area of code in which all identifiers are guaranteed to be unique.

Namespaces are used to help avoid issues where two independent pieces of code have naming collisions with each other when used together.

`namespace`

Name Collision Example

If you were told to go to Room 129 BUT not told which building on campus, how would you know which building to choose for Room 129?

Being told NH 129 or ERB 129 makes a big difference in where you end up.

namespace

ERB.h

```
void PrintLocation(int RoomNumber)
{
   std::cout << "ERB " << RoomNumber;
}
```

NH.h

```
void PrintLocation(int RoomNumber)
{
   std::cout << “NH " << RoomNumber;
}
```

namespace

```cpp
// namespace demo using ERB and NH

#include <iostream>

#include "ERB.h"
#include "NH.h"

int main()
{
    int RoomNumber{};

    std::cout << "Enter Room Number ";
    std::cin >> RoomNumber;

    PrintLocation(RoomNumber);

    return 0;
}
```

g++ xxxx.cpp -E


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

Terminal - student@maverick: /media/sf_VM/CSE1325

File Edit View Terminal Tabs Help

```
student@maverick:/media/sf_VM/CSE1325$
```

namespace

ERB.h

```
namespace ERB
{
     void PrintLocation(int RoomNumber)
     {
          cout << "ERB " << RoomNumber;
     }
}
```

NH.h

```
namespace NH
{
     void PrintLocation(int RoomNumber)
     {
          cout << “NH " << RoomNumber;
     }
}
```

Terminal - student@maverick: /media/sf_VM/CSE1325

File Edit View Terminal Tabs Help

```
student@maverick:/media/sf_VM/CSE1325$ make
g++ -c -g -std=c++11 namespaceERBNHDemo.cpp -o namespaceERBNHDemo.o
namespaceERBNHDemo.cpp: In function ‘int main()’:
namespaceERBNHDemo.cpp:15:2: error: ‘PrintLocation’ was not declared in this sco
pe
   15 |  PrintLocation(RoomNumber);
      |  ^~~~~~~~~~~~~
namespaceERBNHDemo.cpp:15:2: note: suggested alternatives:
In file included from namespaceERBNHDemo.cpp:5:
ERB.h:3:7: note:   ‘ERB::PrintLocation’
    3 |  void PrintLocation(int RoomNumber)
      |       ^~~~~~~~~~~~~
In file included from namespaceERBNHDemo.cpp:6:
NH.h:3:7: note:   ‘NH::PrintLocation’
    3 |  void PrintLocation(int RoomNumber)
      |       ^~~~~~~~~~~~~
make: *** [makefile:14: namespaceERBNHDemo.o] Error 1
student@maverick:/media/sf_VM/CSE1325@
```

Terminal - student@maverick: /media/sf_VM/CSE1325

File Edit View Terminal Tabs Help

```
student@maverick:/media/sf_VM/CSE1325$ m
```

Scope Resolution Operator

You can tell the compiler to look at a particular namespace by using the scope resolution operator

::

with the name of the namespace.

```
ERB::PrintLocation(RoomNumber);
NH::PrintLocation(RoomNumber);
```

So what is

`std::cout`

actually doing?

Scope Resolution Operator

```cpp
#include <iostream>

using namespace std;

#include "ERB.h"
#include "NH.h"
using namespace ERB;
int main()
{
    int RoomNumber{};

    cout << "Enter Room Number ";
    cin >> RoomNumber;

    PrintLocation(RoomNumber);
    NH::PrintLocation(RoomNumber);

    return 0;
}
```

Explicit Type Conversion – Casting
C vs C++

Explicit cast in C

```
int i1 = 10;
int i2 = 4;
float f = (float)i1 / i2;
```

C-style explicit cast in C++ syntax

```
int i1 = 10;
int i2 = 4;
float f = float(i1) / i2;
```

Because these types of casts are not checked by the compiler, they can be misused.

C++ introduced compile-time type checking; therefore, making type casting safer.

Explicit Type Conversion – Casting C++

type value is being converted to

value being converted

```
static_cast

int i1 = 10;
int i2 = 4;
float f = static_cast<float>(i1) / i2;
```

`static_cast` takes a single value as input and outputs the same value converted to the type specified inside the angled brackets.

`static_cast` is best used to convert one fundamental type into another.

Enumerated Types

One of the simplest user-defined data type is the enumerated type.

An **enumerated type** (also called an **enumeration** or **enum**) is a data type where every possible value is defined as a symbolic constant (called an **enumerator**).

Enumerations are defined via the **enum** keyword.

Enumerated Types

```
enum Color
{
    COLOR_BLACK,
    COLOR_RED,
    COLOR_BLUE,
    COLOR_GREEN,
    COLOR_WHITE,
    COLOR_CYAN,
    COLOR_YELLOW,
    COLOR_MAGENTA
};
```

```
Color Banana = COLOR_YELLOW;

Color Blueberry{COLOR_BLUE};

Color Celery(COLOR_GREEN);
```

Providing a name for an enumeration is optional.

Enumerated Types

Like a struct in C – no memory allocation until a variable is declared.

Defining an enumeration does not allocate any memory. When a variable of the enumerated type is defined, memory is allocated for that variable at that time.

Note that each enumerator is separated by a comma and the entire enumeration ends with a semicolon.

Enumerated Types

Unintended side effects can occur when we use multiple enumerations within the same program scope.

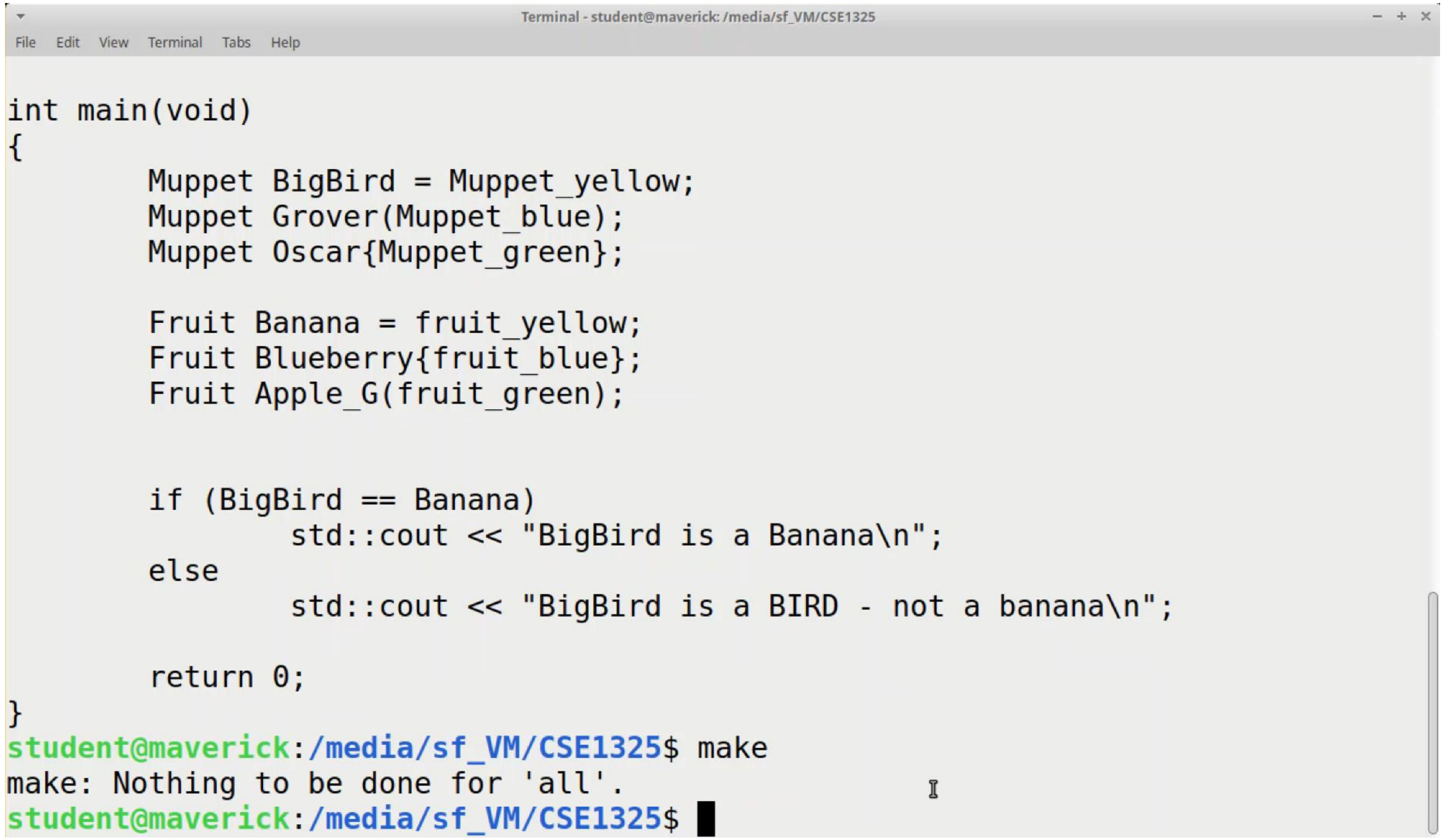
```
enum Muppet
{
   Muppet_red,
   Muppet_yellow,
   Muppet_blue,
   Muppet_green
};
```

```
enum Fruit
{
   fruit_red,
   fruit_yellow,
   fruit_blue,
   fruit_green
};
```

Enumerated Types

```
Muppet BigBird = Muppet_yellow;
Muppet Grover(Muppet_blue);
Muppet Oscar{Muppet_green};


Fruit Banana = fruit_yellow;
Fruit Blueberry{fruit_blue};
Fruit Apple_G(fruit_green);
```




```
if (BigBird == Banana)
   std::cout << "BigBird is a Banana";
```

Enumerated Types

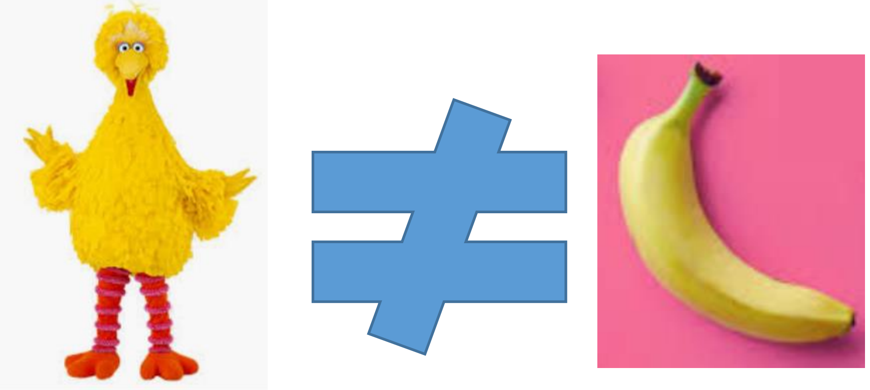
When C++ compares two enumerations (`BigBird` and `Banana`), it implicitly converts both of them to integers and compares the integers.

Since `BigBird` and `Banana` were both been set to enumerators that evaluate to 1, this code will logically assume that `BigBird` is equal to `Banana`.

With standard enumerators, there's no way to prevent comparing enumerators from different enumerations. We get a warning, but it still created an executable.

Since this was probably not our intention,

we have a problem.



Another issue :

Because enumerators are placed into the same namespace as the enumeration, an enumerator name can't be used in multiple enumerations within the same namespace:

```
enum Color
{
        RED,
        BLUE,
        GREEN
};

enum Feeling
{
        HAPPY,
        TIRED,
        BLUE        Line 28
};

student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 enum1Demo.cpp -o enum1Demo.o
enum1Demo.cpp:28:2: error: redeclaration of ‘BLUE’
  BLUE
  ^~~~
```

enum class/scoped enumeration

enum class (also called a **scoped enumeration**) makes enumerations both strongly typed and strongly scoped.

To make an `enum class`, we use the keyword **class** after the enum keyword

```
enum Color
{
     RED,
     BLUE
};

Color Apple = RED;
```

```
enum class Color
{
     RED,
     BLUE
};

Color Apple = Color::RED
```

enum class/scoped enumeration

```
enum Color
{
        RED,
        BLUE,
        GREEN
};

enum Feeling
{
        HAPPY,
        TIRED,
        BLUE
};

student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 enum1Demo.cpp -o
enum1Demo.cpp:28:2: error: redeclarat
  BLUE
  ^~~~
```

```
enum class Color
{
        RED,
        BLUE,
        GREEN
};

enum class Feeling
{
        HAPPY,
        TIRED,
        BLUE
};

student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 enum1Demo.cpp -o enum1Demo.o
g++ -g -std=c++11 enum1Demo.o -o enum1Demo.e
```

Added class

Added class

scoped enumeration

- user defined type
- keywords `enum class`
- includes a type name and a set of identifiers
- identifiers/enumeration constants must be integer constants
- the value of the enumeration constants start at 0 (unless specified otherwise)
- enumeration constants increment by 1
- identifiers in an `enum class` must be unique

scoped enumeration

$$\overbrace{\texttt{enum class}}^{\text{keyword}}\ \overbrace{\texttt{Status}}^{\text{type name}}\ \overbrace{\texttt{\{CONTINUE, WON, LOST\}}}^{\text{enumeration constants}}\texttt{;}$$

`Status` is the scoped `enum`'s type name

`CONTINUE` has a value of 0, `WON` has a value of 1 and `LOST` has a value of 2.

```
(gdb) ptype Status
type = enum class Status : int {Status::CONTINUE, Status::WON, Status::LOST}
```

scoped enumeration

```
enum class Status {CONTINUE, WON, LOST};
```

`Status` is a type so we can declare a variable of that type.

```
Status gameStatus;
```

Then we can assign the enumerated values to our new variable.

```
gameStatus = Status::LOST;
```

```
(gdb) p gameStatus
$1 = Status::LOST
```

scoped enumeration

```
enum class Status {CONTINUE, WON, LOST};
```

```
gameStatus = Status::LOST;
```

We use the scope resolution operator to "tie" `LOST` to `Status`.

If we leave it off

```
enumclassDemo.cpp: In function 'int main()':
enumclassDemo.cpp:10:15: error: 'LOST' was not declared in this scope
  gameStatus = LOST;
```

```
gameStatus = LOST;
```

scoped enumeration

Using the :: allows other enumerated classes to reuse the same enumerated constants with different values.

```
enum class Player1Status {CONTINUE, WON, LOST};
enum class Player2Status {LOST, CONTINUE, WON};

Player1Status P1Stat;
Player2Status P2Stat;
```

`P1Stat` is a variable of type `Player1Status`
`P2Stat` is a variable of type `Player2Status`

has a value of 2

```
P1Stat = Player1Status::LOST;
P2Stat = Player2Status::LOST;
```

has a value of 0

The `auto` keyword

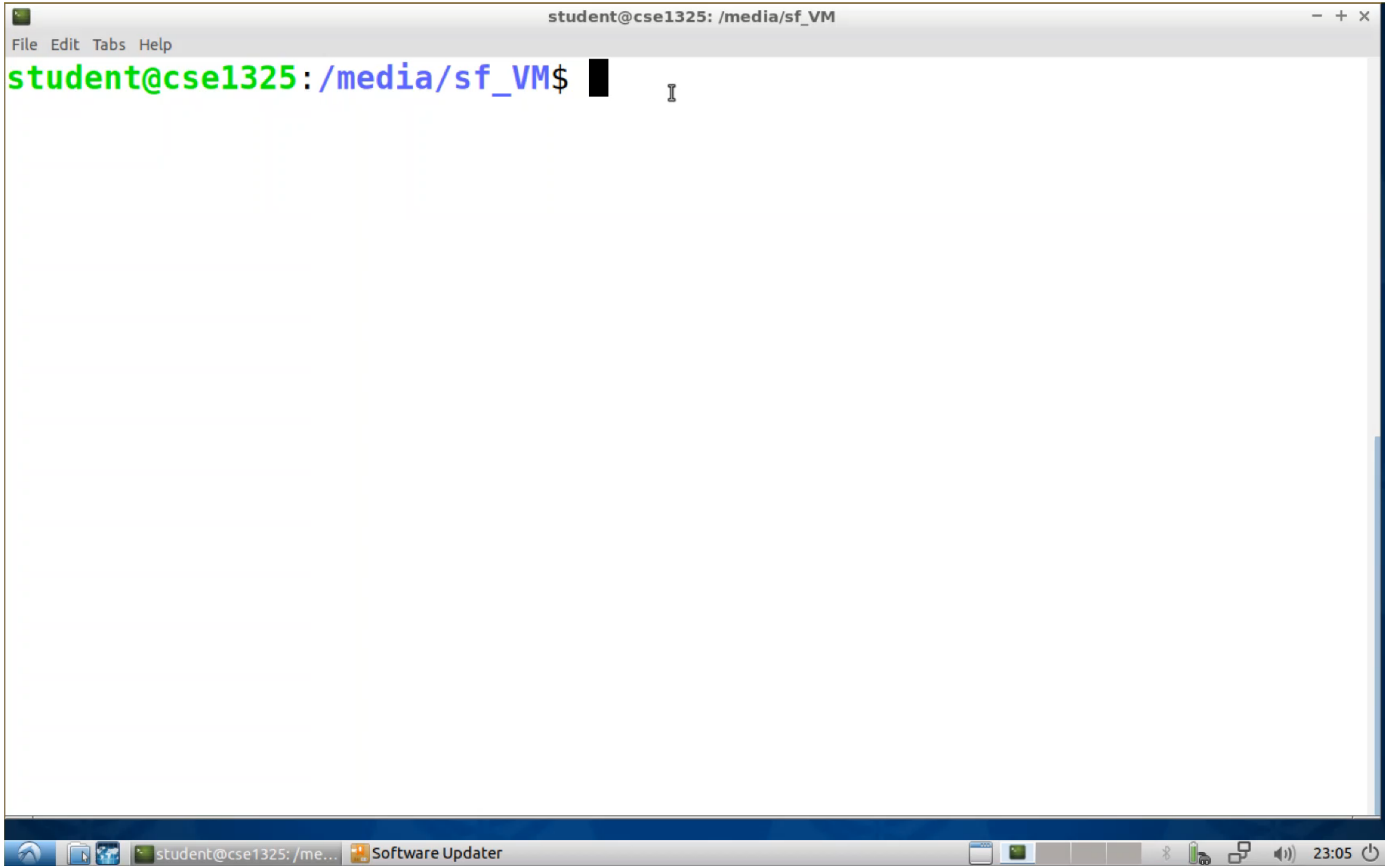
When initializing a variable, the `auto` keyword can be used in place of the variable type to tell the compiler to infer the variable's type from the initializer's type.

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



```
student@cse1325:/media/sf_VM$ more autoDemo.cpp
// auto demo

#include <iostream>

double add(double x, int y)
{
    return x + y;
}

int main()
{
    auto sum = add(5.2, 6);
        std::cout << sum;

    return 0;
}
student@cse1325:/media/sf_VM$
```

```
// auto demo

#include <iostream>

double add(double x, int y)
{
    return x + y;
}
 
int main()
{
    auto sum = add(5.2, 6);
	std::cout << sum;

    return 0;
}
```

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.