10

CSE 1325

Week of 10/26/2020

Instructor : Donna French

```
#include <iostream>

bool CallFunA()
{
   int a = 1, b = 2;

   if (a >= b)
   {
      return true;
   }
   else if (a < b)
   {
      return false;
   }
}

int main(void)
{
   CallFunA();

   return 0;
}
```

```
ifelseDemo.cpp: In function ‘bool CallFunA()’:
ifelseDemo.cpp:17:1: warning: control reaches end
of non-void function [-Wreturn-type]
   17 | }
      | ^
```

Standard Stream Objects

`cin`

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

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

`cout`

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

```
cout << grade;
```

Standard Stream Objects

`cerr`

- `ostream` object
- "connected to" the standard error device (normally the screen)
- uses stream insertion operator `<<`
- outputs to object `cerr` are unbuffered
  - each stream insertion to `cerr` causes its output to appear immediately

```
#include <iostream>

using namespace std;

int main()
{
	cout << "Hello there.  How are you?";
	cerr << "\nNot well - feeling a bit erroritable";
	cout << "Sorry to hear that";

	return 0;
}
```

```
student@maverick:/media/sf_VM/CSE1325$ ./cerrDemo.e
Not well - feeling a bit erroritable.  Hello there.  How are you?  Sorry to hear
 that.  student@maverick:/media/sf_VM/CSE1325$
```

cerrDemo.cpp

```
#include <iostream>

using namespace std;

int main()
{
	cout << "Hello there.  How are you?" << endl;
	cerr << "\nNot well - feeling a bit erroritable. ";
	cout << "Sorry to hear that" << endl;

	return 0;
}
```

```
student@maverick:/media/sf_VM/CSE1325$ ./cerrDemo.e
Hello there.  How are you?
Not well - feeling a bit erroritable.  Sorry to hear that.
student@maverick:/media/sf_VM/CSE1325$
```

cerrDemo.cpp

Standard Stream Objects

`clog`

- `ostream` object
- "connected to" the standard error device (normally the screen)
- uses stream insertion operator `<<`
- outputs to object `clog` are buffered
  - each stream insertion to `clog` is held in an internal memory buffer until the buffer is filled or until the buffer is flushed

```
#include <iostream>

using namespace std;

int main()
{
	cout << "Hello there.  How are you?";
	clog << "Not well - feeling a bit erroritable";
	cout << "Sorry to hear that";

	return 0;
}
```

clogDemo.cpp

```
8             cout << "Hello there.  How are you?  ";
(gdb)
9             clog << "Not well - feeling a bit erroritable.  ";
(gdb)
Not well - feeling a bit erroritable.  10            cout << "Sorry to hear that.
";
(gdb)
12            return 0;
(gdb)
13    }
(gdb)
__libc_start_main (main=0x555555555189 <main()>, argc=1, argv=0x7fffffffe0d8,
    init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out>,
    stack_end=0x7fffffffe0c8) at ../csu/libc-start.c:342
342    ../csu/libc-start.c: No such file or directory.
(gdb)
Hello there.  How are you?  Sorry to hear that.  [Inferior 1 (process 11913) exited
normally]
(gdb)
The program is not being run.
```

scope resolution operator ::

A class's member functions can be defined outside of the class itself by using the scope resolution operator :: to "tie" the function to the class.

```
ClassName::memberFuntionName()
```

The `ClassName::` tells the compiler that the member function is within that class's scope and its name is known to other class members.

```
class CokeMachine
{
   public :
      std::string getMachineName(void)
      {
         return machineName;
      }
};
```

```
CokeMachine.h: In function 'std::__cxx11::string getMachineName()':
CokeMachine.h:137:9: error: 'machineName' was not declared in this scope
  return machineName;
         ^

makefile:17: recipe for target 'Code2_1000074079.o' failed
make: *** [Code2_1000074079.o] Error 1
```

```
class CokeMachine
{
   public :
      std::string getMachineName(void);
};

std::string                    getMachineName(void)
{
   return machineName;
}
```

```
class CokeMachine
{
   public :
      std::string getMachineName(void)
      {
         return machineName;
      }
};
```

```
CokeMachine.h:135:13: error: prototype for 'std::__cxx11::string CokeMachine::ge
tMachineName()' does not match any in class 'CokeMachine'
 std::string CokeMachine::getMachineName(void)
             ^
```

```
class CokeMachin
{
   public :
      std::string getMachineName(int);
};

std::string CokeMachine::getMachineName(void)
{
   return machineName;
}
```

UML Relationships

There are many methods of showing relationships between classes.
We are going to focus on four specific relationships.

Association 

Composition 

Aggregation 

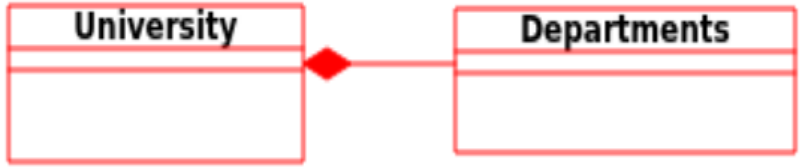
Inheritance 

UML Relationships

Association

- Represents the "has a" relationship
- a linkage between two classes
- shows that classes are aware of each other and their relationship
- uni-directional or bi-directional

UML Relationships



Aggregation

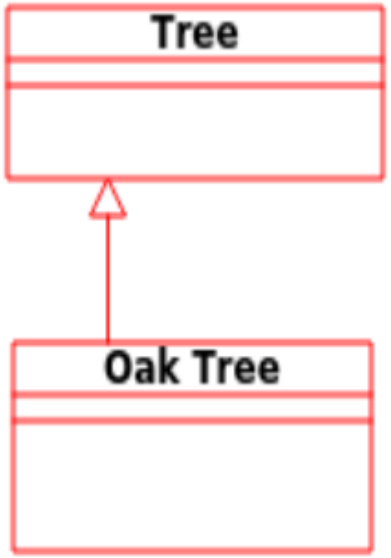
- Special type of association
- Represents the "has a" /"whole-part" relationship.
- Describes when a class (the whole) is composed of/has other classes (the parts)
- Diamond on "whole" side

Composition

- An association that represents a very strong aggregation
- Represents the "has a" /"whole-part" relationship.
- Describes when a class (the whole) is composed of/has other classes (the parts) BUT the parts cannot exist without the whole.
- Diamond on "whole" side

UML Relationships

Inheritance



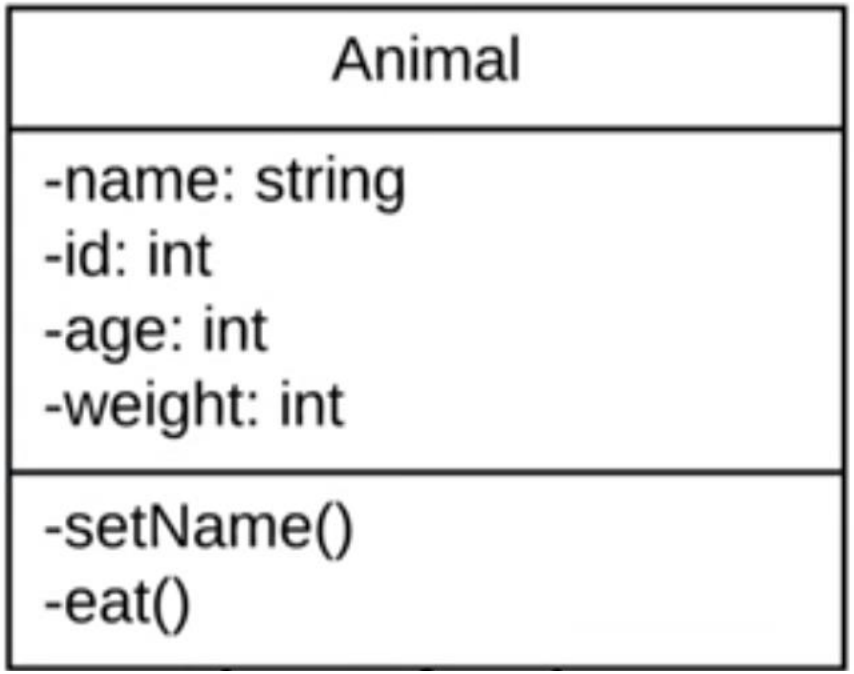
- Represents the "is a" relationship
- Shows the relationship between a super class/base class and a derived/subclass.
- Arrow is on the side of the base class

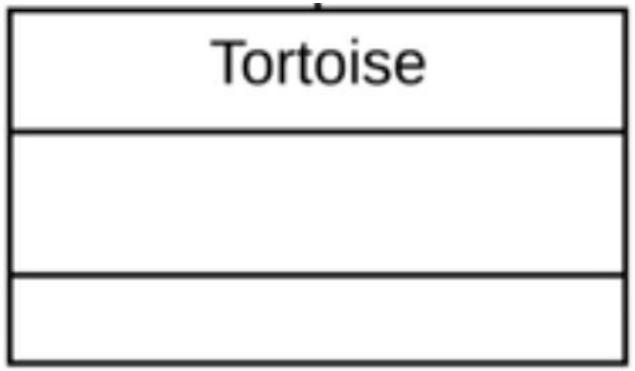
"is a" or "has a"?

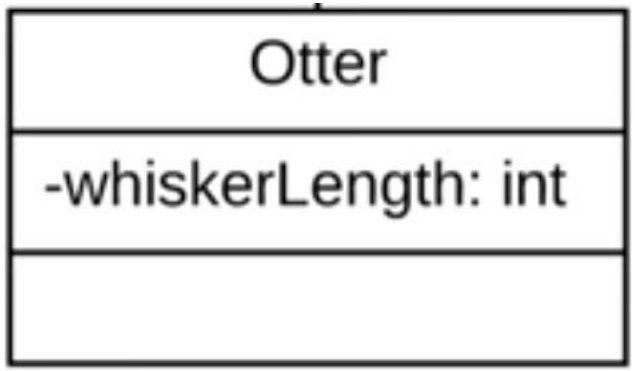
Tortoise is an Animal
Otter is an Animal
Slow Loris is an Animal

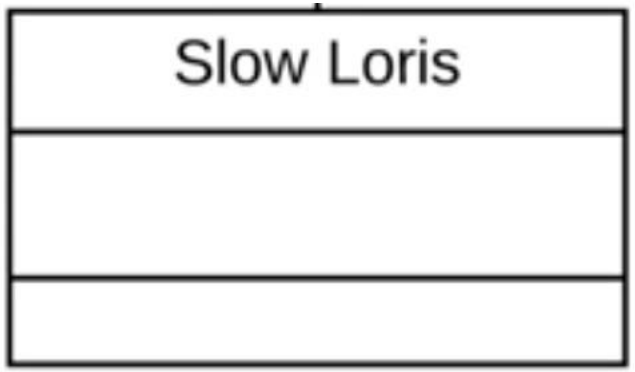
"is a" relationship

Inheritance









"is a" or "has a"?

Otter is a Sea Urchin?
Otter has a Sea Urchin?

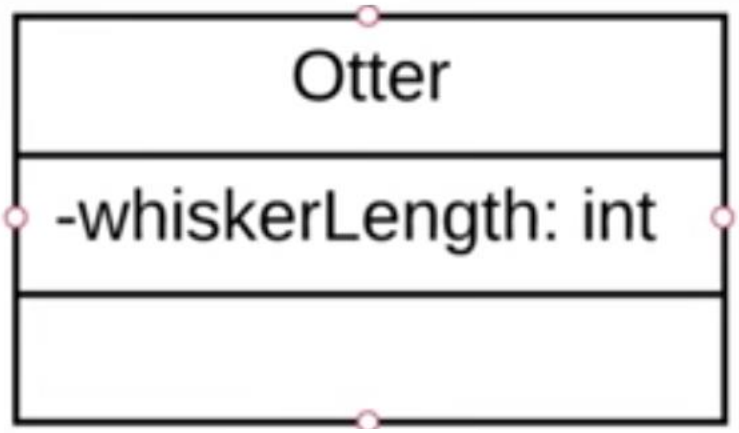
"has a" relationship

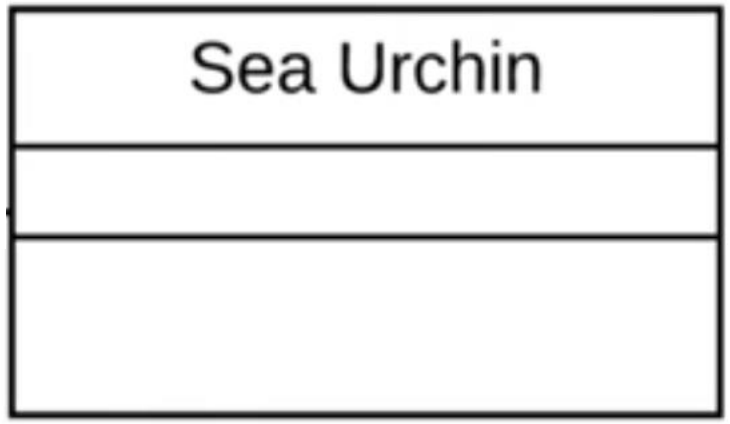
association/composition/aggregation?

"whole/part" relationship?

Otter is part of Sea Urchin?
Sea Urchin is part of Otter?

not "whole/part"





Association

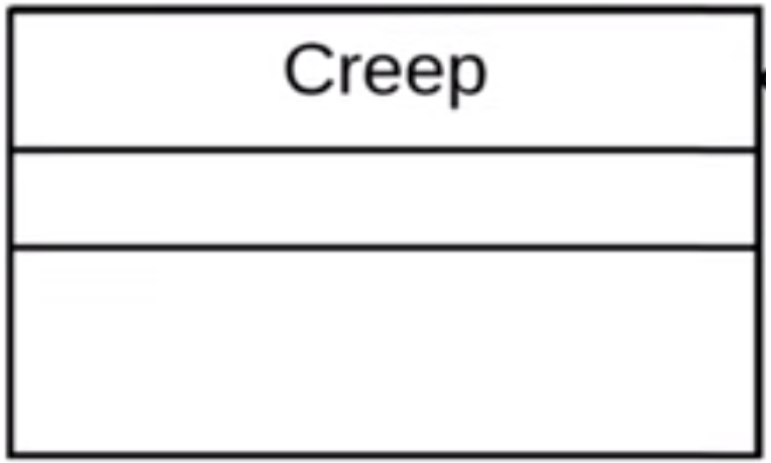
"is a" or "has a"?

Creep is a Tortoise?
Creep has a Tortoise?

"has a" relationship

association/composition/aggregation?

"whole/part" relationship?



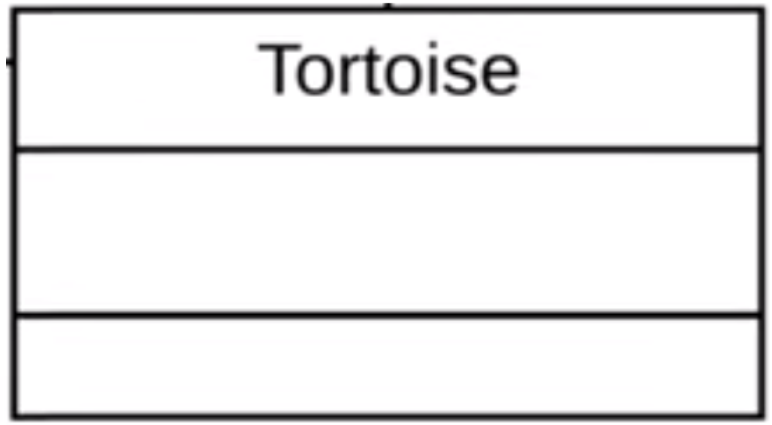
Creep is part of Tortoise?
Tortoise is part of Creep?

Yes – Creep is "whole" and Tortoise is "part"

Can the Creep exist without the Tortoise?

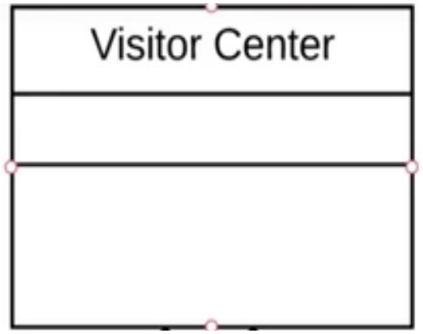
Yes

Aggregration



"is a" or "has a"?

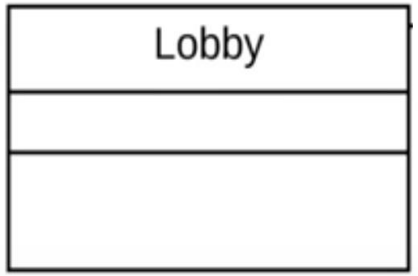
Lobby is a Visitor Center? Bathroom is a Visitor Center?
Visitor Center is a lobby or bathroom?

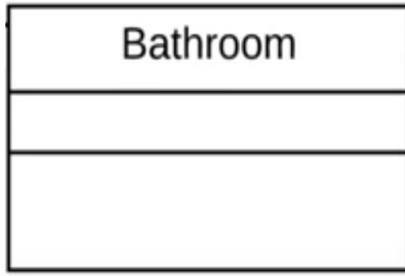


Lobby/Bathroom has a Visitor Center?
Visitor Center has a Lobby and a Bathroom?

"has a" relationship

association/composition/aggregation?





"whole/part" relationship?

Can the "parts" – Bathroom and Lobby – exist without the "whole" - Visitor Center?

Visitor Center is part of Bathroom/Lobby?
Bathroom/Lobby is part of Visitor Center?

Yes – Visitor Center is "whole" and Bathroom/Lobby is "part"

Composition

`namespace`

A program may include many identifiers defined in different scopes.

Sometimes a variable of one scope will collide with a variable of the same name in a different scope which could possibly create a naming conflict.

Identifier overlapping occurs frequently in third-party libraries that happen to use the same names for global identifiers such as function names.

Example - multiple classes using a function named `getName()` and a private data member named `name`.

namespace

C++ solves this conflict with `namespace`.

Each `namespace` defines a scope in which identifiers and variables are placed.

To use a `namespace` member

member's name must be qualified with the `namespace` name and the scope resolution operator (::)

```
MyNameSpace::member
```

using directive must appear before the name is used in the program

```
using namespace MyNameSpace;
```

namespace

To use a `namespace` member

member's name must be qualified with the `namespace` name and the scope resolution operator (::)

```
MyNameSpace::member
```

`using` directive must appear before the name is used in the program

```
using namespace MyNameSpace;
```

```
std::cout << "Hello";
```

```
using namespace std;
cout << "Hello";
```

member's name must be qualified with the `namespace` name and the scope resolution operator (::)

```
#include <iostream>

int main()
{
  std::string name;

  std::cout << "Please enter your name ";

  getline(std::cin, name);

  std::cout << "Your name is "
            << name << std::endl;

  return 0;
}
```

`using` directive must appear before the name is used in the program

```
#include <iostream>

using namespace std;

int main()
{
  string name;

  cout << "Please enter your name ";

  getline(cin, name);

  cout << "Your name is " << name << endl;

  return 0;
}
```

namespace

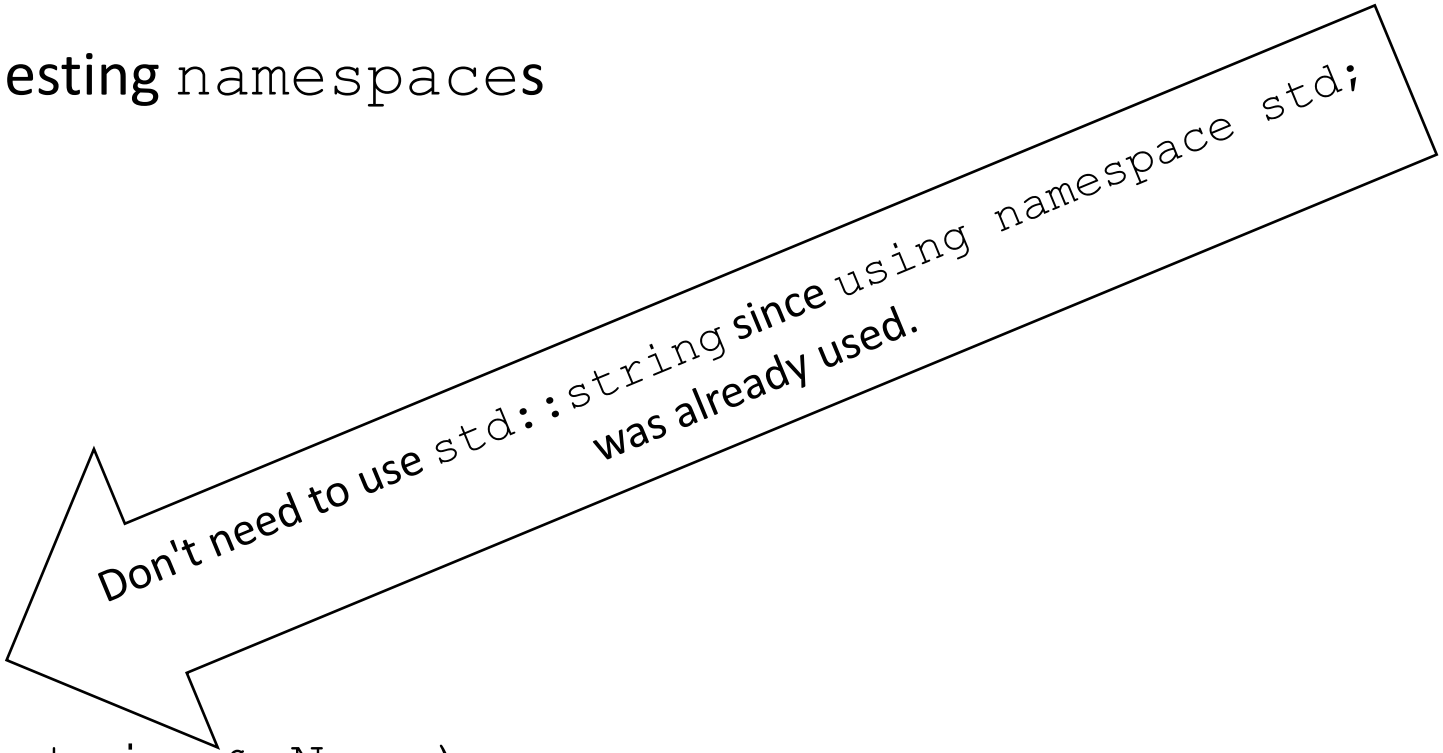
Creating your own `namespace`

```
namespace MySpace
{
  int cin;
  std::string name{"Fred"};

  void getline(int cin, std::string& Name)
  {
    Name = name;
  }
}

using namespace MySpace;
```

namespace

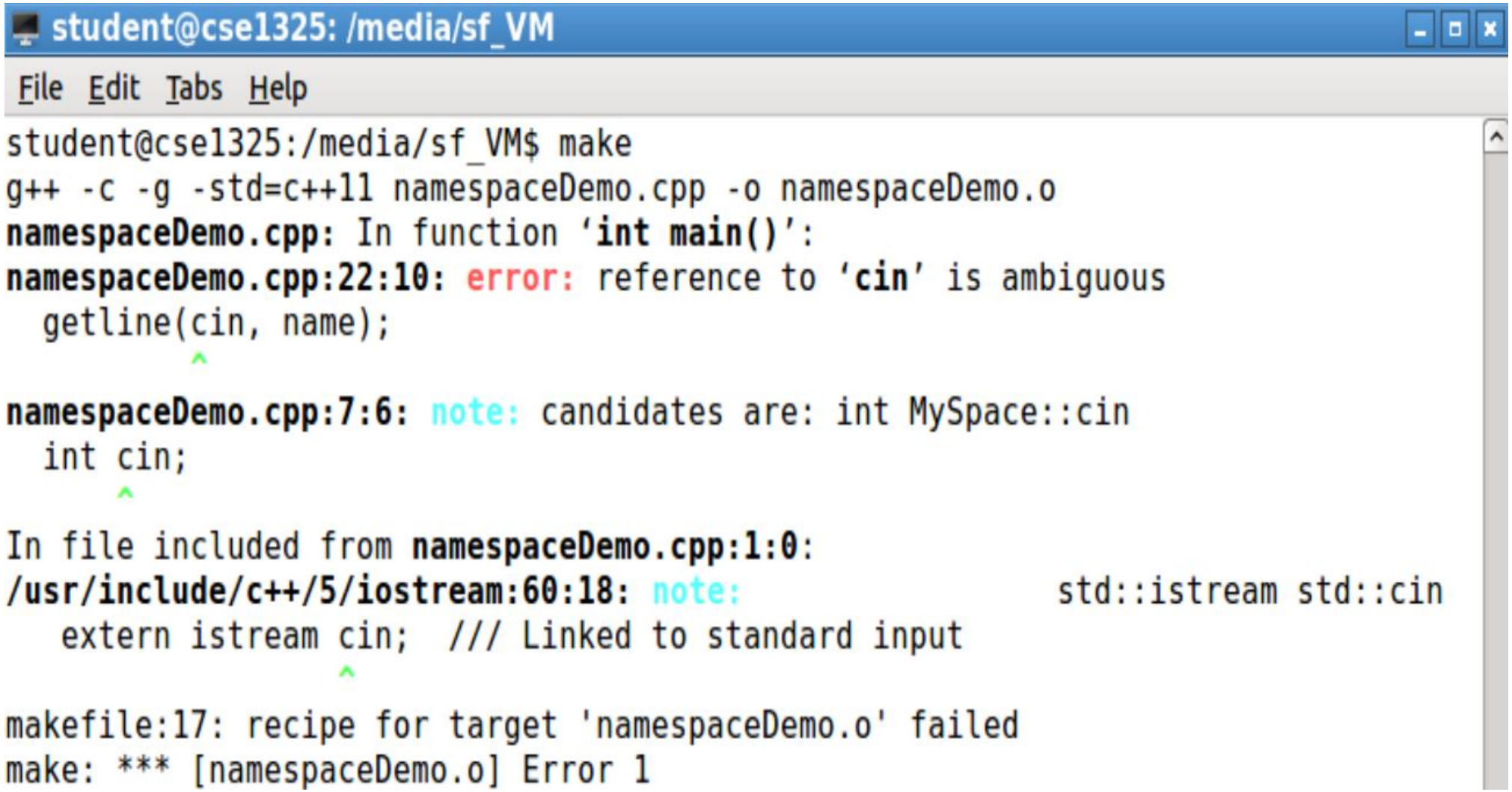
Nesting namespaces

```
using namespace std;

namespace MySpace
{
  int cin;
  string name{"Fred"};

  void getline(int cin, string& Name)
  {
    Name = name;
  }
}

using namespace MySpace;
```



namespace

```
using namespace std;

namespace MySpace
{
  int cin;
  string name{"Fred"};

  void getline(int cin, string& Name)
  {
    Name = name;
  }
}

using namespace MySpace;

int main()
{
  string name;

  cout << "Please enter your name ";

  getline(cin, name);

  cout << "Your name is " << name << endl;

  return 0;
}
```



How do we fix this error?

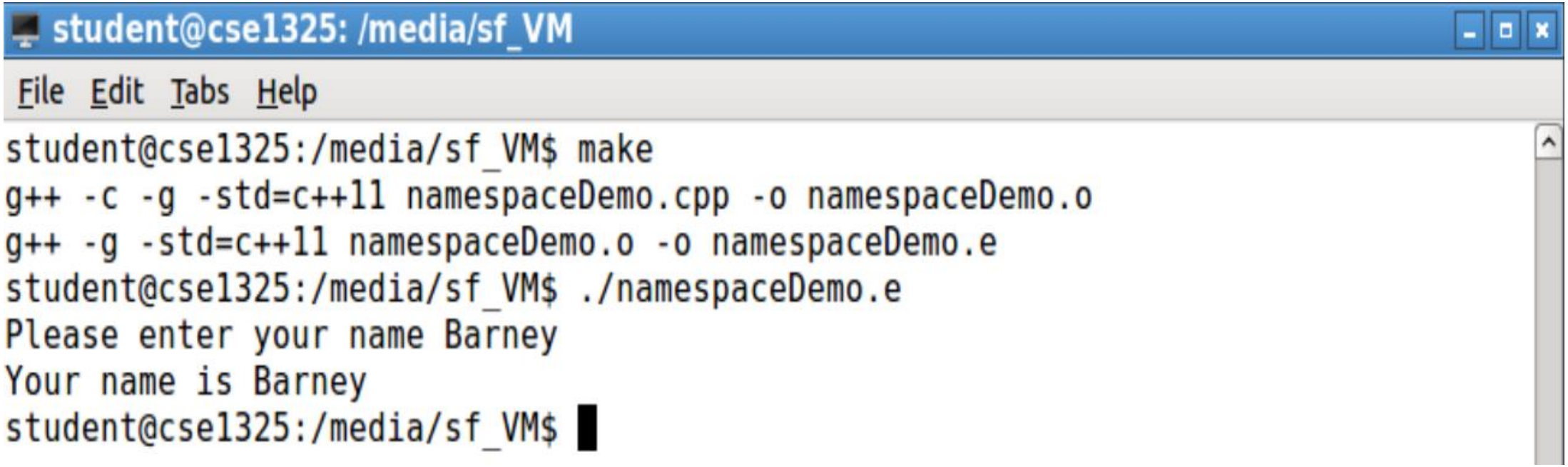
namespace

```
using namespace std;

namespace MySpace
{
  int cin;
  string name{"Fred"};

  void getline(int cin, string& Name)
  {
    Name = name;
  }
}

using namespace MySpace;

int main()
{
  string name;

  cout << "Please enter your name ";

  getline(cin, name);

  cout << "Your name is " << name << endl;

  return 0;
}
```

namespace

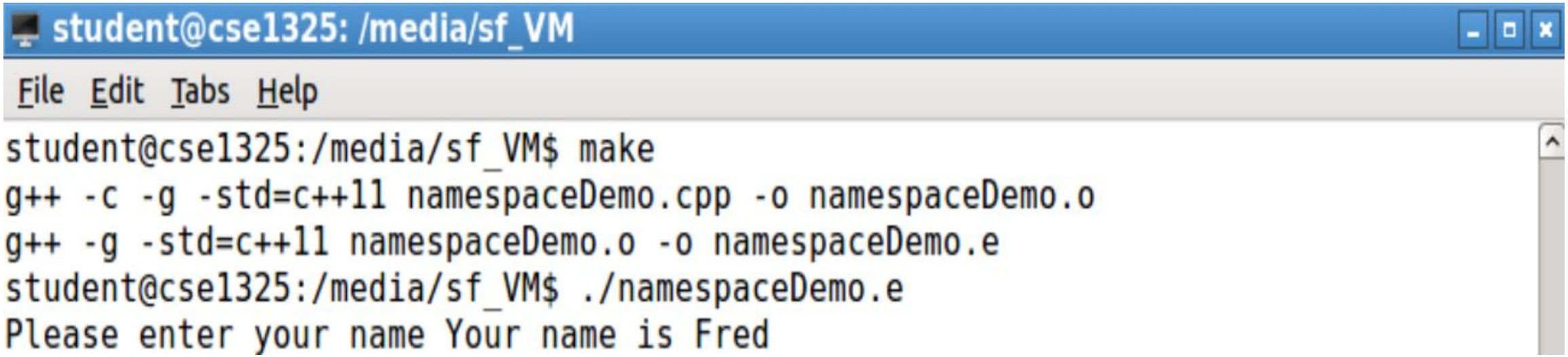
```
getline(cin, name);
```

```
getline(std::cin, name);
```



namespace

```
getline(cin, name);

getline(MySpace::cin, name);
```



namespace

```
using namespace std;

namespace MySpace
{
  int cin;
  string name{"Fred"};

  void getline(int cin, string& Name)
  {
    Name = name;
  }
}

using namespace MySpace;

int main()
{
  string name;

  cout << "Please enter your name ";

  getline(MySpace::cin, name);

  cout << "Your name is " << name << endl;

  return 0;
}
```

```
Please enter your name Your name is Fred
```

namespace

`using namespace` should not be placed in header files

If I defined `MySpace` in a header file and included the usage of

```
using namespace MySpace;
```

and then included that header file in my program, all of my program's usages of `cin` would need to be qualified with `std::` even though I also included `using namespace std;`

Introduction to Exception Handling

An **exception** indicates a problem that occurs while a program executes.

Should be a problem that occurs infrequently; hence; exception.

Exception handling allows you to create fault-tolerant programs that can handle exceptions.

This may mean allowing the program to finish normally even if an exception occurred – like trying to access an out-of-range subscript in a vector.

More severe problems might require that the program notify the user of the problem and then terminate immediately.

Introduction to Exception Handling

When a function detects a problem, like trying to access a subscript out of bounds, it **throws** an exception.

We can see this if we try to if we use member function `at()` to try to access vector elements out of range.

```
vector<int> WholeNumbers={0,1,2,3,4};

for (int i = 0; i <= WholeNumbers.size(); i++)
{
   cout << WholeNumbers.at(i) << endl;
}
```

```
vector<int> WholeNumbers={0,1,2,3,4};

for (int i = 0; i <= WholeNumbers.size(); i++)
{
    cout << WholeNumbers.at(i) << endl;
}
cout << "Even if an exception occurs, life goes on" << endl;
```

What happens when `WholeNumbers.at(5)` trys to print?

```
student@cse1325:/media/sf_VM$ ./trycatchDemo.e
0
1
2
3
4
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
Aborted (core dumped)
```

the program terminated after throwing an error.

```
4
12              for (int i = 0; i <= WholeNumbers.size(); i++)
(gdb)
17                      cout << WholeNumbers.at(i) << endl;
(gdb)
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)

Program received signal SIGABRT, Aborted.
0x00007ffff74ab428 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
54      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb)

Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
(gdb)
```

```
vector<int> WholeNumbers={0,1,2,3,4};

for (int i = 0; i <= WholeNumbers.size(); i++)
{
  try
  {
    cout << WholeNumbers.at(i) << endl;
  }
  catch (out_of_range& ex)
  {
    cerr << "An exception occurred: " << ex.what() << endl;
  }
}

cout << "Even if an exception occurs, life goes on" << endl;
```

`try` block – contains the code that might **throw** an exception

`catch` block – contains the code that handles the exception

Introduction to Exception Handling

```
catch (out_of_range& ex)
{
   cerr << "An exception occurred: " << ex.what() << endl;
}
```

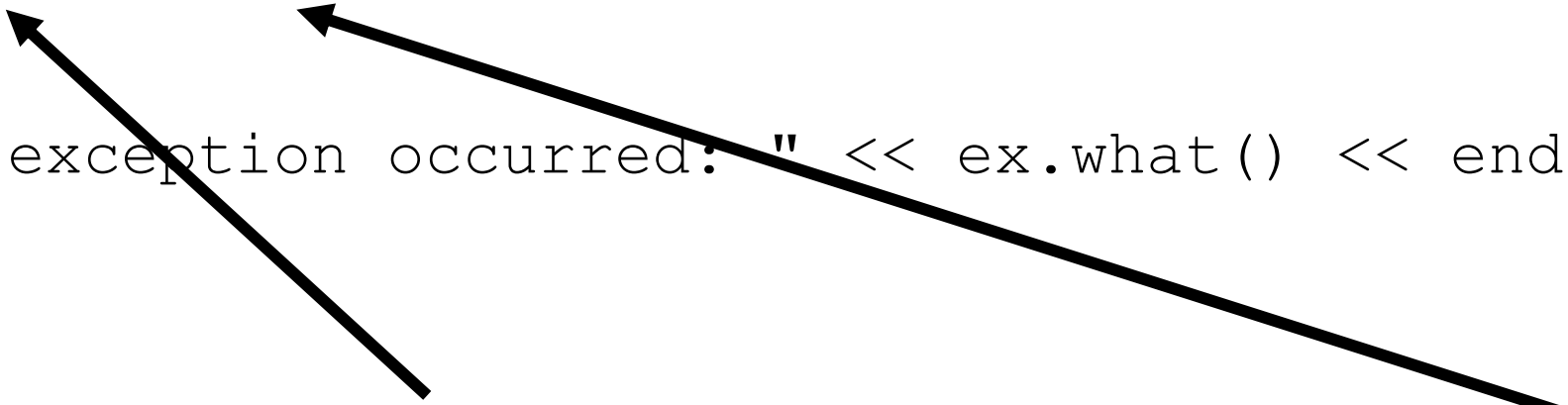
Accessing an out of range `vector` element triggers the `vector` member function `at()` to throw an exception of `out_of_range`.

When `at()` throws the exception, the code in the `try` block terminates immediately and the code in the `catch` block begins executing.

Any variables declared in a `try` block are out of scope and not accessible in the `catch` block.

Introduction to Exception Handling

```
catch (out_of_range& ex)
{
    cerr << "An exception occurred: " << ex.what() << endl;
}
```



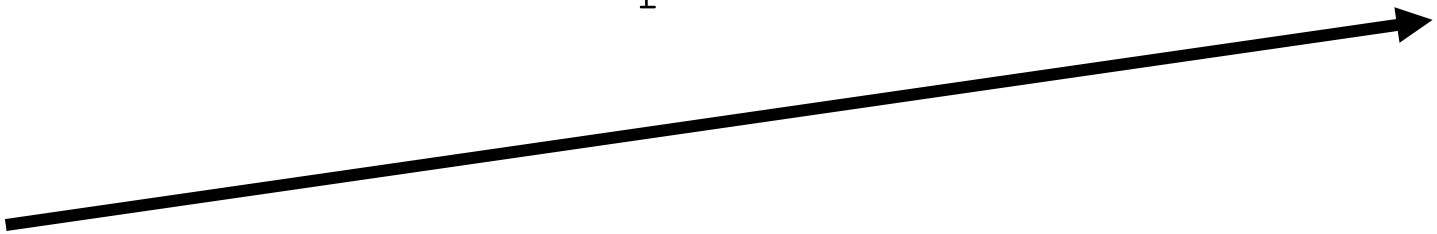
The catch block declares a type (`out_of_range`) and an exception parameter (`ex`) that it receives as a reference.

`ex` is the caught exception object

- catching an exception by reference increases performance by preventing the exception object from being copied when it is caught

Introduction to Exception Handling

```
catch (out_of_range& ex)
{
    cerr << "An exception occurred: " << ex.what() << endl;
}
```



`what()` is a member function of the exception object

`what()` will get the error message that is stored in the exception object and display it

Once the message is displayed, the exception is considered handled and the program continues with the next statement after the `catch` block's closing brace.

```
vector<int> WholeNumbers={0,1,2,3,4};

for (int i = 0; i <= WholeNumbers.size(); i++)
{
  try
  {
    cout << WholeNumbers.at(i) << endl;
  }
  catch (out_of_range& ex)
  {
    cerr << "An exception occurred: " << ex.what() << endl;
  }
}

cout << "Even if an exception occurs, life goes on" << endl;
```

```
student@cse1325:/media/sf_VM$ ./trycatchDemo.e
0
1
2
3
4
An exception occurred: vector::_M_range_check: __n (which is 5) >= this->size()
(which is 5)
Even if an exception occurs, life goes on
```

The `try` block passes control to the `catch` block when `vector`'s member function `at()` throws an exception.

The `catch` block uses the exception object's `what()` member function to print out the exception and allows the program to continue and print the statement immediately after the `catch` block and the program ends normally.

In debug, we can see that the program exited normally.

```
4
13              for (int i = 0; i <= WholeNumbers.size(); i++)
(gdb)
17                              cout << WholeNumbers.at(i) << endl;
(gdb)
An exception occurred: vector::_M_range_check: __n (which is 5) >= this->size()
(which is 5)
Even if an exception occurs, life goes on
[Inferior 1 (process 3462) exited normally]
```

```
(gdb)
std::vector<int, std::allocator<int> >::_M_range_check (this=0x7fffffffe0c0,
    __n=5) at /usr/include/c++/5/bits/stl_vector.h:802
802             if (__n >= this->size())
(gdb)
std::vector<int, std::allocator<int> >::size (this=0x7fffffffe0c0)
    at /usr/include/c++/5/bits/stl_vector.h:655
655           { return size_type(this->_M_impl._M_finish - this->_M_impl._M_star
t); }
(gdb)
std::vector<int, std::allocator<int> >::_M_range_check (this=0x7fffffffe0c0,
    __n=5) at /usr/include/c++/5/bits/stl_vector.h:803
803               __throw_out_of_range_fmt(__N("vector::_M_range_check: __n "
(gdb)
std::vector<int, std::allocator<int> >::size (this=0x7fffffffe0c0)
    at /usr/include/c++/5/bits/stl_vector.h:655
655           { return size_type(this->_M_impl._M_finish - this->_M_impl._M_star
t); }
(gdb)
An exception occurred: vector::_M_range_check: __n (which is 5) >= this->size()
(which is 5)
Even if an exception occurs, life goes on
[Inferior 1 (process 3466) exited normally]
```

Separating Interface from Implementation

Each of our prior class definition examples put the class in a header file for reuse and then included the header in a source code file containing `main()`.

`CokeMachine.h` `Code2_1000074079.c`

This arrangement of files allowed us to create and manipulate objects of the class.

This arrangement also reveals the entire implementation of the class to the class's clients since a header file is a text file that can be opened and read.

Separating Interface from Implementation

The client code actually should only know 3 things about a class

- what member functions to call
- what arguments to provide to each member function
- what return type to expect from each member function

The client code does not need to know how those functions are implemented.

Separating Interface from Implementation

When the client code does know how a class is implemented, then the programmer might write client code based on the class's implementation details.

Ideally, if the class's implementation changes, the class's clients should not have to change.

Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully, eliminating changes to the client code.

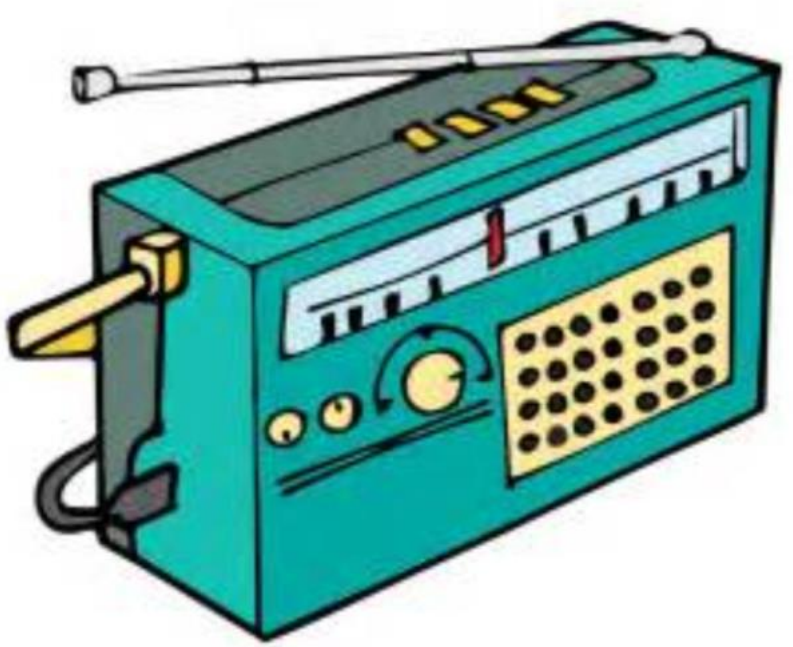
Interface of a Class

We are all familiar with a radio, and, in general, the controls that allow us to perform a limited set of operations on the radio – changing the station, adjusting the volume and choosing between AM and FM.

Some radios use dials, some have push buttons and some support voice commands.

These controls serve as the interface between the radio's users and the internal components.

The interface specifies what operations a radio permits users to perform but does not specify how the operations are implemented inside the radio.



Interface of a Class

The interface of a class describes *what* services a class's clients can use and how to *request* those services, but not *how* the class carries out the services.

A class's `public` interface consists of the class's `public` member functions which can also be known as the class's `public` services.

We can specify a class's interface by writing a class definition that lists only the class's member function prototypes and the class's data members.

Separating the Interface from the Implementation

To separate the class's interface from the implementation, we break up the class into two files – the header in which the class is defined and the source code file in which the class's member functions are defined.

By convention, member function definitions are placed in a source code file of the same base name as the class's header file but with a .cpp filename extension.

Doing so allows the following…

1. the class is still reusable
2. the clients of the class know what member functions the class provides, how to call them and what return types to expect
3. the clients do not know how the class's member functions are implemented

```
// Time.h

#include <string>

class Time
{
   public:


   private:
      unsigned int hour{0};   // 0 - 23 (24-hour clock format)
      unsigned int minute{0}; // 0 - 59
      unsigned int second{0}; // 0 - 59
};
```

```
// set new Time value using universal time
void setTime(int h, int m, int s)
{
  // validate hour, minute and second
  if ((h >= 0 && h < 24) &&
      (m >= 0 && m < 60) &&
      (s >= 0 && s < 60))
  {
    hour = h;
    minute = m;
    second = s;
  }
  else
  {
    throw invalid_argument("hour, minute and/or second was out of range");
  }
}
```

```
// return Time as a string in universal-time format (HH:MM:SS)
std::string toUniversalString() const
{
  ostringstream output;
  output << setfill('0') << setw(2) << hour << ":"
         << setw(2) << minute << ":" << setw(2) << second;
  return output.str(); // returns the formatted string
}

// return Time as string in standard-time format (HH:MM:SS AM or PM)
std::string toStandardString() const
{
  ostringstream output;
  output << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
         << setfill('0') << setw(2) << minute << ":" << setw(2)
         << second << (hour < 12 ? " AM" : " PM");
  return output.str(); // returns the formatted string
}
```

```
// Time.h

#include <string>


// include guard
#ifndef TIME_H
#define TIME_H


class Time
{
   public:
      void setTime(int, int, int);           // set hour, minute and second
      std::string toUniversalString() const; // 24-hour time format string
      std::string toStandardString() const;  // 12-hour time format string
   private:
      unsigned int hour{0};   // 0 - 23 (24-hour clock format)
      unsigned int minute{0}; // 0 - 59
      unsigned int second{0}; // 0 - 59
};
#endif
```

Instead of function definitions, the class contains function prototypes that describe the class's public interface without revealing the member function implementation.

We don't have a constructor in this example but it would be included here as well.

This is enough information for the compiler to create an object (reserve enough memory) and ensure that it is called properly.

Include Guard

When we build larger programs, other definitions and declarations will also be placed in the headers.

The include guard prevents the code between the `#ifndef` and `#endif` from being `#include`d if the name `TIME_H` has been defined.

When `Time.h` is `#include`d the first time, the identifier `TIME_H` is not yet defined. In this case, the `#define` directive defines `TIME_H` and the preprocessor includes the `Time.h` header's contents in the `.cpp` file.

If the header is `#include`d again, `TIME_H` is defined already and the code in between `#ifndef` and `#endif` is ignored by the preprocessor.

Time.cpp

```
#include <iomanip> // for setw and setfill stream manipulators
#include <stdexcept> // for invalid_argument exception class
#include <sstream> // for ostringstream class
#include <string>
#include "Time.h" // include definition of class Time from Time.h
```

```
// set new Time value using universal time
void Time::setTime(int h, int m, int s)
{
  // validate hour, minute and second
  if ((h >= 0 && h < 24) &&
      (m >= 0 && m < 60) &&
      (s >= 0 && s < 60))
  {
    hour = h;
    minute = m;
    second = s;
  }
  else
  {
    throw invalid_argument("hour, minute and/or second was out of range");
  }
}
```

throws an exception of type `invalid_argument` (from `<stdexcept>`). The `throw` statement creates a new object of type `invalid_argument`. The custom message in the "" is passed to `invalid_argument`'s contstructor

Time.cpp

Time.cpp

```
// return Time as a string in universal-time format (HH:MM:SS)
std::string Time::toUniversalString() const
{
  ostringstream output;
  output << setfill('0') << setw(2) << hour << ":"
         << setw(2) << minute << ":" << setw(2) << second;
  return output.str(); // returns the formatted string
}

// return Time as string in standard-time format (HH:MM:SS AM or PM)
std::string Time::toStandardString() const
{
  ostringstream output;
  output << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
         << setfill('0') << setw(2) << minute << ":" << setw(2)
         << second << (hour < 12 ? " AM" : " PM");
  return output.str(); // returns the formatted string
}
```

`setfill(n)`

sticky manipulator

specifies a fill character that is displayed when an integer is output in a field wider than the number of digits in the value

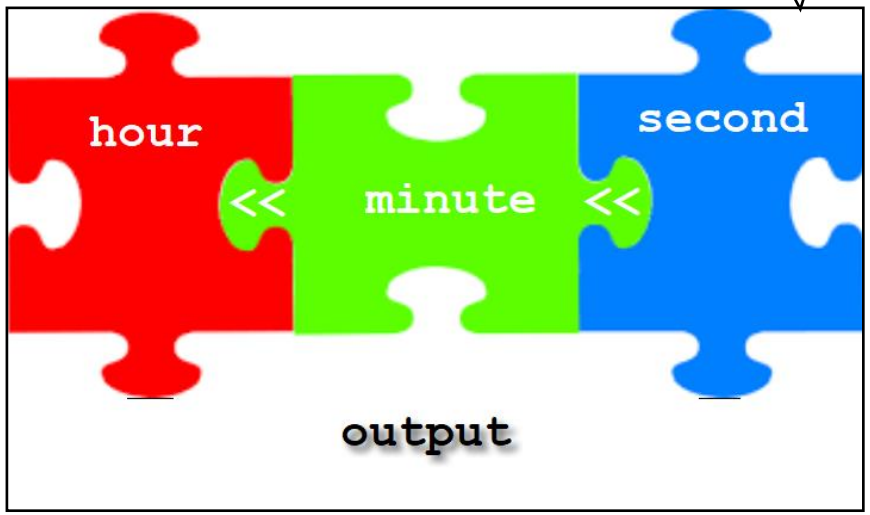
the fill characters appear to the left of the digits in the number because the number is right aligned by default

So 2 becomes 02

```
std::string Time::toUniversalString() const
{
  ostringstream output;
```

Why const?

from header <sstream>




```
  output << setfill('0') << setw(2) << hour << ":"
         << setw(2) << minute << ":" << setw(2) << second;
```

`setw()` is not sticky and `setfill()` is

Put the puzzle pieces of `hour`, `minute` and `second` together into 1 string – `output`.

```
  return output.str(); // returns the formatted string
}
```

`str()` is a member function of `ostringstream` that returns the string

```
// TestTime.cpp

#include <iostream>
#include <stdexcept> // for invalid_argument exception class
#include "Time.h" // include definition of class Time from Time.h

using namespace std;

// displays a Time in 24-hour and 12-hour formats
void displayTime(const string& message, const Time& time)
{
   cout << message << "\nUniversal time: " << time.toUniversalString()
        << "\nStandard time: " << time.toStandardString() << "\n";
}
```

global function

message passed by reference but marked as const. Why?

time passed by reference but marked as const. Why?

member function of Time

member function of Time

TimeTest.cpp

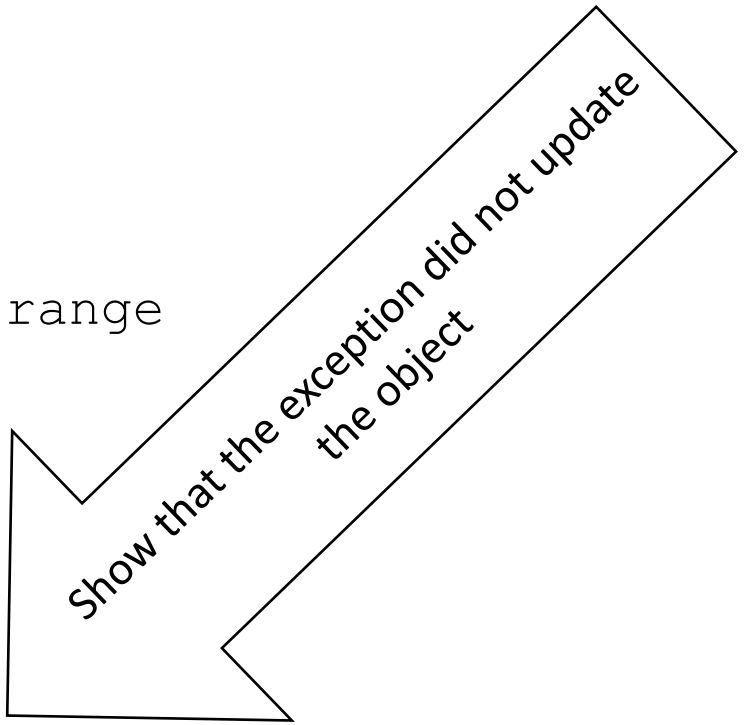
Global vs Member Functions

Member functions `toUniversalString()` and `toStandardString()` member functions take no arguments because they implicitly know that they are to create string representations of the data for a particular Time object on which they are invoked.

Using an object-oriented programming approach often requires fewer arguments when calling functions.

Also reduces the likelyhood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

```
int main(void)
{
   Time MyTime;

   displayTime("Initial time:", MyTime);
   MyTime.setTime(13, 27, 6); // change time
   displayTime("After setTime:", MyTime);

   // attempt to set the time with invalid values
   try
   {
      MyTime.setTime(99, 99, 99); // all values out of range
   }
   catch (invalid_argument& say)
   {
      cout << "Exception: " << say.what() << "\n\n";
   }

   // display time value after attempting to set an invalid time
   displayTime("After attempting to set an invalid time:", MyTime);
}
```



TimeTest.cpp

```
Initial time:
Universal time: 00:00:00
Standard time: 12:00:00 AM

After setTime:
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting to set an invalid time:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Compiling and Linking

Now we have three files

`Time.h` – header file with class definition and function prototypes

`Time.cpp` – member function code

`TimeTest.cpp` – a program to create Time objects and test them

How do we compile these into one executable?

Compiling and Linking

Seems like objects would be quite large because they contain data member and member functions. Instantiating multiple objects from a class would; therefore, take up a lot of space.

Objects only contain data so objects are much smaller than if they also contained member functions.

The compiler creates only one copy of the member functions separate from all objects of the class.

All objects of the class share this one copy.

makefile for two modules and header

```
SRC1 = TimeTest.cpp
SRC2 = Time.cpp
OBJ1 = $(SRC1:.cpp=.o)
OBJ2 = $(SRC2:.cpp=.o)
EXE = $(SRC1:.cpp=.e)



CFLAGS = -g -std=c++11

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2)
     g++ $(CFLAGS) $(OBJ1) $(OBJ2) -o $(EXE)

$(OBJ1) : $(SRC1)
     g++ -c $(CFLAGS) $(SRC1) -o $(OBJ1)

$(OBJ2) : $(SRC2)
     g++ -c $(CFLAGS) $(SRC2) -o $(OBJ2)
```