CSE 1325

Week of 10/19/2020

Instructor : Donna French

`this`

One of the questions about classes often asked is,

“When a member function is called, how does C++ keep track of which object it was called on?”.

The answer is that C++ utilizes a hidden pointer named “this”!

`this`

There's only one copy of each class's functionality, but there can be many objects of a class, so how do member functions know which object's data members to manipulate?

Every object has access to its own address through a pointer called `this` (a C++ keyword).

The `this` pointer is not part of the object itself. The memory occupied by the `this` pointer is not reflected in the result of a `sizeof` operation on the object. Rather, the `this` pointer is passed (by the compiler) as an implicit argument to each of the object's non-static member functions.

this

```
class Simple
{
   public:
      Simple(int id)
      {
         setID(id);
      }
      void setID(int id)
      {
         m_id = id;
      }
      int getID()
      {
         return m_id;
      }
    private:
      int m_id;
};
```

```
int main()
{
    Simple simple(1);
    simple.setID(2);
    std::cout << simple.getID();

    return 0;
}
```

What would this print?

2

`this`

When we call

```
simple.setID(2);
```

C++ knows that function `setID()` should operate on object `simple` and that `m_id` actually refers to `simple.m_id`.

How?

this

Let's look at this line of code

```
simple.setID(2);
```

Although the call to function `setID()` looks like it only has one argument, it actually has two!

When compiled, the compiler converts `simple.setID(2);` into the following

```
setID(&simple, 2);
```

Note that `simple` has been changed from an object prefix to a function argument!

this

```
setID(&simple, 2);
```

`setID()` is now just a standard function call, and the object `simple` (which was formerly an object prefix) is now passed by address as an argument to the function.

Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter.

this

Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter.

```
void setID(int id)
{
   m_id = id;
}
```

is converted by the compiler into

```
void setID(Simple* const this, int id)
{
   this->m_id = id;
}
```

this

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

When the compiler compiles a normal member function, it **implicitly** adds a new parameter to the function named `this`.

The `this` pointer is a hidden `const` pointer that holds the address of the object the member function was called on.

this

```
void setID(Simple* const this, int id)
{
    this->m_id = id;
}
```

Inside the member function, any class members (functions and variables) also need to be updated so they refer to the object the member function was called on.

This is done by adding a `this->` prefix to each of them.

In the body of function `setID()`,`m_id` (which is a class member variable) has been converted to `this->m_id`.

When `this` points to the address of `simple`, `this->m_id` will resolve to `simple.m_id`.

this

When we call

```
      simple.setID(2);
```

the compiler actually calls

```
      setID(&simple, 2);
```

Inside `setID()`, the `this` pointer holds the address of object `simple`.

Any member variables inside `setID()` are prefixed with `this->`.

So when we say `m_id = id`, the compiler is actually executing

```
      this->m_id = id
```

which in this case updates `simple.m_id to id`.

Using the `this` Pointer to Avoid Naming Collisions

Member functions use the `this` pointer

implicitly (as we’ve done so far)

or

explicitly

to reference an object’s data members and other member functions. A common explicit use of the `this` pointer is to avoid naming conflicts between a class’s data members and member-function parameters (or other local variables)

`this`

All of this happens automatically.

Just remember is that all normal member functions have a `this` pointer that refers to the object the function was called on

`this` always points to the object being operated on.

So how many “this” pointers exist?

Each member function has a `this` pointer parameter that is set to the address of the object being operated on.

this

```
int main()
{
    Simple A(1); // this = &A inside the Simple constructor
    Simple B(2); // this = &B inside the Simple constructor
    A.setID(3);  // this = &A inside member function setID
    B.setID(4);  // this = &B inside member function setID

    return 0;
}
```

The `this` pointer alternately holds the address of object A or B depending on whether we've called a member function on object A or B.

`this` is just a function parameter - it doesn't add any memory usage to your class

Using the `this` Pointer to Avoid Naming Collisions

We are familiar with the implicit call

```
class Test
{
   public :
      void print(void) const
      {
        cout << "x = " << x;
      }
   private :
      int x{0};
};
```

Compiler is translating print() to
`void print(&A) const`
to implicitly pass the pointer to the object

Using the `this` Pointer to Avoid Naming Collisions

```
void setHour(int hour)
{
    if (hour >= 0 && hour < 24)
    {
            hour = hour;
    }
}
```

`hour` is passed into the function `setHour` and then validation is performed on it.

If it passes validation, then we want to update the object's data member `hour`.

We do this by using `this`.

Using the `this` Pointer to Avoid Naming Collisions

So why not just use different names?

```
void setHour(int hourA)
{
    if (hourA >= 0 && hourA < 24)
    {
        this->hour = hourA;
    }
}
```

A widely accepted practice to minimize the proliferation of identifier names is to use the same name for a set function’s parameter and the data member it sets, and to reference the data member in the set function’s body via this->.

Using the `this` Pointer to Avoid Naming Collisions

We are familiar with the implicit usage so what does the explicit usage look like?

```
class Test
{
   public :
      void print(void) const
      {
         cout << "x = " << (*this).x;
         cout << "x = " << this->x;
      }
   private :
      int x{0};
};
```

`this`

Recommendation

Do not add `this->` to all uses of your class members.

Only do so when you have a specific reason to.

We will see more examples of when using `this` is necessary.

Type of `this` pointer

The type of the `this` pointer depends on the type of the object and whether the member function in which this is used is declared `const`

In a non-`const` member function of class `Employee`, the `this` pointer has the type

```
Employee* const
```

a constant pointer to a nonconstant `Employee`.

In a `const` member function, this has the type

```
const Employee* const
```

a constant pointer to a constant `Employee`.

`this` is a `const` pointer -- you can change the value of the underlying object it points to, but you can not make it point to something else.

Operator Overloading

>>

- stream extraction operator
- bitwise right shift operator

<<

- stream insertion operator
- bitwise left shift operator

These are familiar operators that are overloaded.

Operator Overloading

+ and –

Each of these performs differently depending on their context

integer addition
floating point arithmetic
pointer arithmetic

These are familiar operators that are overloaded meaning that the compiler generates the appropriate code based on the types of the operands.

Operator Overloading

The C++ Standard Library's class `string` has lots of overloaded operators.

```
string s1{"happy"};
string s2{"birthday"};
string s3;

cout << "s1\t" << s1 << "\ns2\t" << s2 << "\ns3" << s3 << endl;
```

```
s1   happy
s2   birthday
s3
```

stringoverloadDemo.cpp

```
string s1{"happy"};
string s2{"birthday"};
```

Operator Overloading

```
cout << "\ns2 == s1\t" << (s2 == s1)
     << "\ns2 != s1\t" << (s2 != s1)
      << "\ns2 >  s1\t" << (s2 > s1)
      << "\ns2 <  s1\t" << (s2 < s1)
      << endl;
```

```
s2 == s1      false
s2 != s1      true
s2 >  s1      false
s2 <  s1      true
```

stringoverloadDemo.cpp

```
string s1{"happy"};
string s2{"birthday"};
string s3;
```

Operator Overloading

```
cout << "\n\ns3 = " << s3 << endl;
s3 = s1;
cout << "\ns3 = " << s3 << endl;

s3[0] = 'H';
s2[0] = 'B';
s3 += s2;
cout << "\n\n" << s3 << endl;
```

```
s3 =
s3 = happy

HappyBirthday
```

stringoverloadDemo.cpp

Operator Overloading

The operators that have been overloaded for `string`s provide a concise notation for manipulating those `string` objects.

We can overload operators with our own user-defined types as well.

C++ does not allow new operators to be created but it does allow most existing operators to be overloaded so that when they are used with objects, they have special meanings.

Operator Overloading

Most of C++'s operators can be overloaded.

There are a few exceptions

- `.`
- `.*` (pointer to member)
- `::`
- `?:`

Operator Overloading
Rules and Restrictions

- An operator's precedence cannot be changed by overloading
  - () can be used to force the order of evaluation of overloaded operators
- An operator's associativity cannot be changed by overloading
  - if an operator normally associates from left to right then so will it when overloaded
- An operator's "ary"ness cannot be changed
  - overloaded unary operators remain unary
  - overloaded binary operators remain binary
  - ternary operator ?: cannot be overloaded (C++'s only ternary operator)
- On existing operators can be overloaded
  - you cannot create new operators
- You cannot overload operators to change how the operator works on fundamental types
  - cannot make the + operator do subtraction
- Related operators, like + and +=, must be overloaded separately
- When overloading (), [], -> or any assignment operators, the operator overload function must be declared as a class member. All other overloaded operators can be member functions

Operator Overloading

In C++, operators are implemented as functions.

By using function overloading on the operator functions, you can define your own versions of the operators that work with different data types (including classes that you’ve written).

Using function overloading to overload operators is called operator overloading.

Operator Overloading

```
int x = 2;
int y = 3;
std::cout << x + y << '\n';
```

Think of + as a function that adds two values and returns the result.

```
operator+(x,y)
```

and `cout` prints the return value

Operator Overloading

```
Mystring string1 = "Hello, ";
Mystring string2 = "World!";
std::cout << string1 + string2 << '\n';
```

Does this concatenate `string1` and `string2` to make

```
Hello, World!
```

No, because `string1` and `string2` are of objects instantiated from class `Mystring`.

Why? Because + is not defined for class `Mystring`.

Operator Overloading

When evaluating an expression containing an operator, the compiler uses the following rules

- If *all* of the operands are fundamental data types, the compiler will call a built-in routine if one exists. If one does not exist, the compiler will produce a compiler error.
- If *any* of the operands are user data types, the compiler looks to see whether the type has a matching overloaded operator function that it can call. If it can't find one, it will try to convert one or more of the user-defined type operands into fundamental data types so it can use a matching built-in operator. If that fails, then it will produce a compile error.

Overloading Binary Operator

A binary operator can be overloaded as

- a member function with one parameter
- a non-member function with two parameters
  - one parameter must be either a class object or a reference to a class object

Non-member functions are often declared as a friend of a class to access private data

Binary Overload Operators as Member Functions

When would we use < or > or = (for example) to compare two objects?

…when two objects have an attribute (or a combination of attributes) that makes one object "greater than"/"less than" another object.

The attribute(s) that makes one object < or > or = to another object is defined by you the programmer.

So how do we use < to compare two objects?

```
int main()
{
   Quarterback Cowboys{"Dak", 54, 35, 330, 1};
   Quarterback Texans{"Deshaun", 66, 39, 486, 3};

   if (Texans < Cowboys)
      cout << "YES - Texans < Cowboys" << endl;
   else
      cout << "NO - Texans not < Cowboys" << endl;

   return 0;
}


class Quarterback
{
   public :
      Quarterback(std::string name, int att, int comp, int yds, int td)
      : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
      {}



};
```

bomfDemo.cpp

```cpp
// binary overload member function Demo
#include <iostream>

class Quarterback
{
     public :
          Quarterback(std::string name, int att, int comp, int yds, int td)
          : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
          {}

          bool operator<(const Quarterback& QB)
          {
               std::cout << "Is " << this->qbName
                         << " < " << QB.qbName << std::endl;

               if (qbAtt  < QB.qbAtt  &&
                    qbComp < QB.qbComp &&
                    qbYds  < QB.qbYds  &&
                    qbTd   < QB.qbTd)
                    return true;
               else
                    return false;
          }
     private :
          std::string qbName;
          int qbAtt;
          int qbComp;
          int qbYds;
          int qbTd;
};

using namespace std;

int main()
{
     Quarterback Cowboys{"Dak", 54, 35, 330, 1};
     Quarterback Texans{"Deshaun", 66, 39, 486, 3};

     if (Texans < Cowboys)
          cout << "YES - Texans < Cowboys" << endl;
     else
          cout << "NO - Texans not < Cowboys" << endl;

     return 0;
}
```

bomfDemo.cpp

```
class Quarterback
{
   public :
      Quarterback(std::string name, int att, int comp, int yds, int td)
      : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
      {}



   private :
      std::string qbName;
      int qbAtt;
      int qbComp;
      int qbYds;
      int qbTd;
};
```

bomfDemo.cpp

```
class Quarterback
{
   public :
      Quarterback(std::string name, int att, int comp, int yds, int td)
      : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
      {}

      bool operator<(Quarterback& QB)
      {
         std::cout << "Is " << this->qbName
                   << " < " << QB.qbName << std::endl;

         if (qbAtt  < QB.qbAtt  &&
            qbComp < QB.qbComp &&
            qbYds  < QB.qbYds  &&
            qbTd   < QB.qbTd)
            return true;
         else
            return false;
      }
   private :
      std::string qbName;
      int qbAtt;
      int qbComp;
      int qbYds;
      int qbTd;
};
```

bomfDemo.cpp

```
Line 1     bool operator<(const Quarterback& QB)
Line 2     {
Line 3        std::cout << "Is " << this->qbName
Line 4                  << " < " << QB.qbName << std::endl;
Line 5
Line 6        if (qbAtt  < QB.qbAtt  &&
Line 7            qbComp < QB.qbComp &&
Line 8            qbYds  < QB.qbYds  &&
Line 9            qbTd   < QB.qbTd)
Line 10          return true;
Line 11       else
Line 12          return false;
Line 13    }
```

Line 1 Overloading the < operator. Syntax is the word "`operator`" and the operator symbol with no space between
Returns `bool` which is `true` or `false`
Object `QB` is being passed by reference (`Quarterback&`) so we need to use `const` to make sure the object is not changed.

Line 3 The explicit use of `this->` is not required but it is used here for illustration and clarity.

Line 6-9 `this->` is being used implicitly

Line 10 Return `true` if all conditions are true

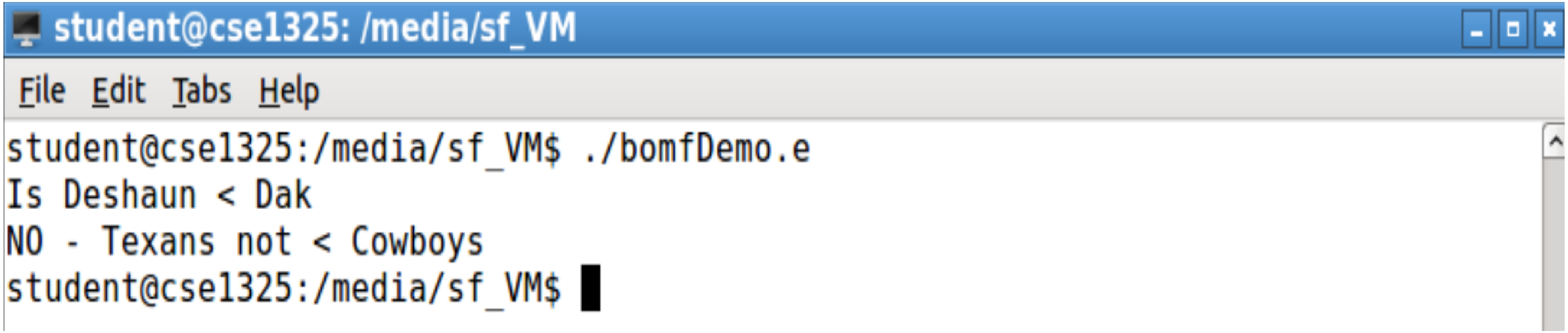
Line 11 Return `false` if all conditions are false

bomfDemo.cpp

```
bool operator<(const Quarterback& QB)
{
  std::cout << "Is " << this->qbName
            << " < " << QB.qbName << std::endl;

  if (qbAtt  < QB.qbAtt  &&
      qbComp < QB.qbComp &&
      qbYds  < QB.qbYds  &&
      qbTd   < QB.qbTd)
    return true;
  else
    return false;
}

Quarterback Cowboys{"Dak", 54, 35, 330, 1};
Quarterback Texans{"Deshaun", 66, 39, 486, 3};

if (Texans < Cowboys)
  cout << "YES - Texans < Cowboys" << endl;
else
  cout << "NO - Texans not < Cowboys" << endl;
```

bomfDemo.cpp

```
Quarterback Cowboys{"Dak", 54, 35, 330, 1};
Quarterback Texans{"Deshaun", 66, 39, 486, 3};

if (Texans < Cowboys)
   cout << "YES - Texans < Cowboys" << endl;
else
   cout << "NO - Texans not < Cowboys" << endl;
```



```
student@cse1325:/media/sf_VM$ ./bomfDemo.e
Is Deshaun < Dak
NO - Texans not < Cowboys
student@cse1325:/media/sf_VM$ 
```

bomfDemo.cpp

Binary Overload Operators as Non-Member Functions

What if we moved member function out of the class and defined it as just a function in the program?

A binary operator can be overloaded as

a member function with one parameter

a non-member function with two parameters

one parameter must be either a class object or a reference to a class object

As a member function with one parameter

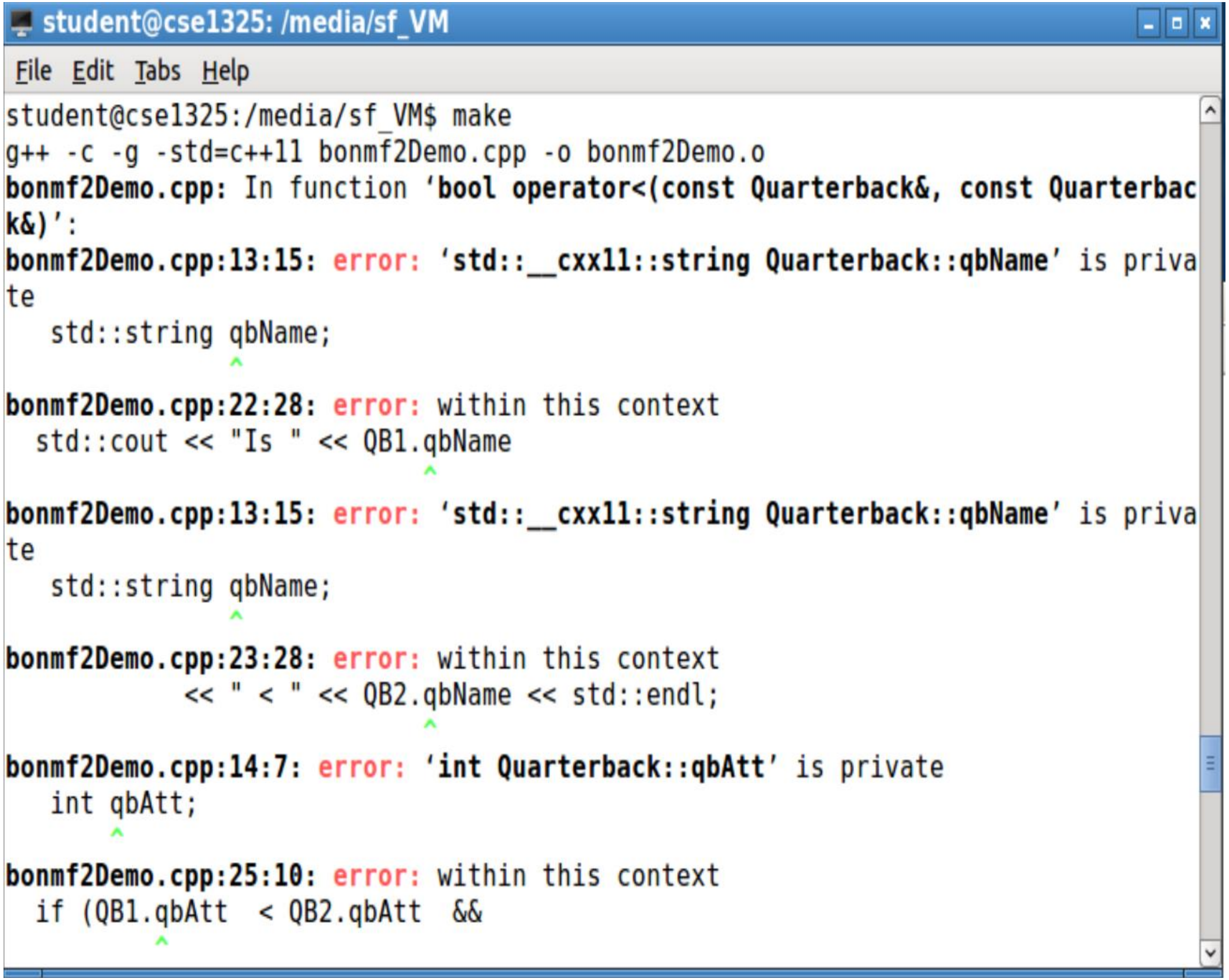
```
bool operator<(const Quarterback& QB)
{
  std::cout << "Is " << this->qbName
            << " < " << QB.qbName << std::endl;

  if (qbAtt  < QB.qbAtt  &&
    qbComp < QB.qbComp &&
    qbYds  < QB.qbYds  &&
    qbTd   < QB.qbTd)
    return true;
  else
    return false;
}
```

As a non member function with two parameters

```
bool operator<(const Quarterback& QB1, const Quarterback& QB2)
{
  std::cout << "Is " << QB1.qbName
            << " < " << QB2.qbName << std::endl;

  if (QB1.qbAtt  < QB2.qbAtt  &&
      QB1.qbComp < QB2.qbComp &&
      QB1.qbYds  < QB2.qbYds  &&
      QB1.qbTd   < QB2.qbTd)
      return true;
  else
      return false;
}
```

What happens when I make this function a non member function?

It is asking to access access private member data…..



```
student@cse1325:/media/sf_VM$ make
g++ -c -g -std=c++11 bonmf2Demo.cpp -o bonmf2Demo.o
bonmf2Demo.cpp: In function ‘bool operator<(const Quarterback&, const Quarterbac
k&)’:
bonmf2Demo.cpp:13:15: error: ‘std::__cxx11::string Quarterback::qbName’ is priva
te
   std::string qbName;
               ^
bonmf2Demo.cpp:22:28: error: within this context
  std::cout << "Is " << QB1.qbName
                            ^
bonmf2Demo.cpp:13:15: error: ‘std::__cxx11::string Quarterback::qbName’ is priva
te
   std::string qbName;
               ^
bonmf2Demo.cpp:23:28: error: within this context
            << " < " << QB2.qbName << std::endl;
                            ^
bonmf2Demo.cpp:14:7: error: ‘int Quarterback::qbAtt’ is private
   int qbAtt;
       ^
bonmf2Demo.cpp:25:10: error: within this context
  if (QB1.qbAtt  < QB2.qbAtt  &&
          ^
```

How do we fix this?

We make a friend

```
class Quarterback
{
  friend bool operator<(const Quarterback& QB1, const Quarterback& QB2);

  public :
    Quarterback(std::string name, int att, int comp, int yds, int td)
    : qbName{name}, qbAtt{att}, qbComp{comp}, qbYds{yds}, qbTd{td}
    {}


  private :
    std::string qbName;
    int qbAtt;
    int qbComp;
    int qbYds;
    int qbTd;
};
```

cin >> setw(x) >> MyString;

```
up to date

up-to-date
```

```
string part1;
string part2;
string part3;

cin >> setw(2) >> part1;
cin.ignore(1);
cin >> setw(2) >> part2;
cin.ignore(1);
cin >> setw(4) >> part3;

cout << part1 << "-" << part2 << "-" << part3 << endl;
part1 = part1 + "-" + part2 + "-" + part3 + "\n";
cout << part1;
```

Overloading >> and <<

```
Enter phone number in the form (555) 555-5555:
(219) 772-2387

The phone number entered was
Area code: 219
Exchange : 772
Line     : 2387
(219) 772-2387
```

PhoneNumber.cpp

```
Enter phone number in the form (555) 555-5555:
(219) 772-2387

The phone number entered was
Area code: 219
Exchange : 772
Line     : 2387
(219) 772-2387
```

```
int main()
{
  PhoneNumber phone;

  cout << "Enter phone number in the form (555) 555-5555:" << endl;

  cin >> phone;

  cout << "\nThe phone number entered was\n";

  cout << phone << endl;

  return 0;
}
```

PhoneNumber.cpp

```
istream& ignore (streamsize n = 1, int delim = EOF);
```

```
cin >> phone;
(219) 772-2387
```

Overloading >> and <<

```
istream& operator>>(istream& input, PhoneNumber& number)
{
     input.ignore();
     input >> setw(3) >> number.areaCode;
     input.ignore(2);  // skip ) and space
     input >> setw(3) >> number.exchange;
     input.ignore();   // skip dash
     input >> setw(4) >> number.line;
     return input;
}
```

PhoneNumber.cpp

Overloading >> and <<

```
ostream& operator<<(ostream& output, const PhoneNumber& number)
{
   output << "Area code: "   << number.areaCode
          << "\nExchange : " << number.exchange
          << "\nLine     : " << number.line << "\n"
          << "(" << number.areaCode << ") "
          << number.exchange
          << "-" << number.line << "\n";
   return output;
}
```

```
cin >> phone;
Area code: 219
Exchange : 772
Line     : 2387
(219) 772-2387
```

Overloading a Binary Operator

```
Quarterback Cowboys{"Dak", 54, 35, 330, 1};
Quarterback Texans{"Deshaun", 66, 39, 486, 3};
```

overloaded member function

Is Deshaun(Texan) < Dak(Cowboy)?

```
if (Texans < Cowboys)
```

```
Texans.operator<(Cowboys)
```

```
bool operator<(const
Quarterback& QB)
```

`this->` references Texans object

overloaded `friend` function

Is Deshaun(Texan) < Dak(Cowboy)?

```
if (Texans < Cowboys)
```

```
operator<(Texans, Cowboys)
```

```
bool operator<(const
Quarterback& QB1, const
Quarterback& QB2)
```

no `this` available or needed

Overloaded Operators as Non-Member `friend` Functions

The normal use of `cin` and `cout` is

```
    cin >> x;
    cout << x;
```

`cin` and `cout` are on the left side of the stream operator.

Therefore, when we overload >> and <<, we want our object to appear on the right side of the operator.

Overloaded Operators as Non-Member `friend` Functions

Overload operator functions for binary operators can be member functions if the left operand is an object of the class in which the function is a member.

So, if the overload functions were member functions, the syntax would be

```
x << cin;            x.operator<<cin;
x >> cout;           x.operator>>cout;
```

This syntax, while correct, could be mistaken as incorrect since it is not the "normal"/"expected" syntax.

Overloaded Operators as Non-Member `friend` Functions

To not use this syntax,

```
x << cin;              x.operator<<cin;
x >> cout;             x.operator>>cout;
```

we make our overload function a friend function instead of a member function and we can use put the object on the right.

```
cin >> x;
cout << x;
```

Operator Overloading

Overloading the relational operators

The test that determines if one object is less than or greater than another object is determined by the programmer. Those tests are written into the operator overload function.

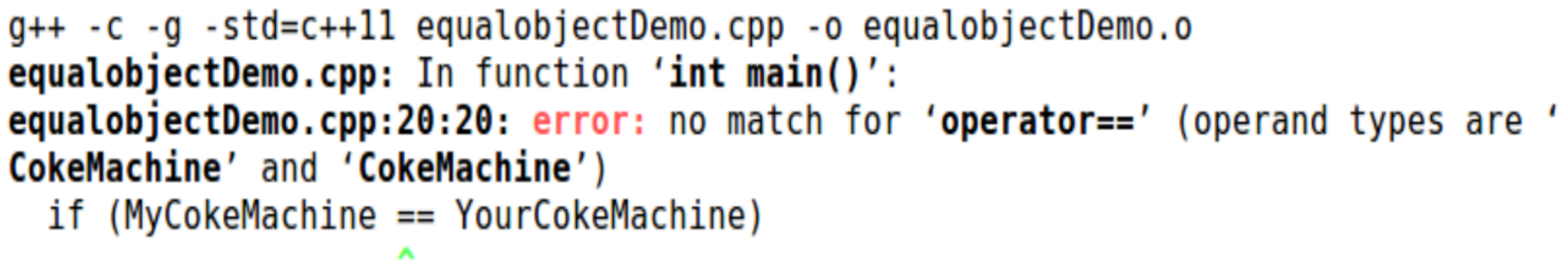
Overloading the stream insertion/extraction operators

<< and >> can be overloaded to accept input or print output based on rules defined by the programmer. Those test are written into the operator overload function.

Operator Overloading

Overloading the == operator can be used to determine if two objects are equal but the definition of equal is still determined by the programmer. Using a non overloaded == will not determine if two objects are equal.

```
CokeMachine MyCokeMachine{"Bob's Coke Machine", 50, 500, 100};
CokeMachine YourCokeMachine{"Bob's Coke Machine", 50, 500, 100};
if (MyCokeMachine == YourCokeMachine)
   cout >> "They are equivalent" << endl;
else
   cout >> "They are not equal" << endl;
```



```
g++ -c -g -std=c++11 equalobjectDemo.cpp -o equalobjectDemo.o
equalobjectDemo.cpp: In function ‘int main()’:
equalobjectDemo.cpp:20:20: error: no match for ‘operator==’ (operand types are ‘
CokeMachine’ and ‘CokeMachine’)
  if (MyCokeMachine == YourCokeMachine)
                    ^
```

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 <<

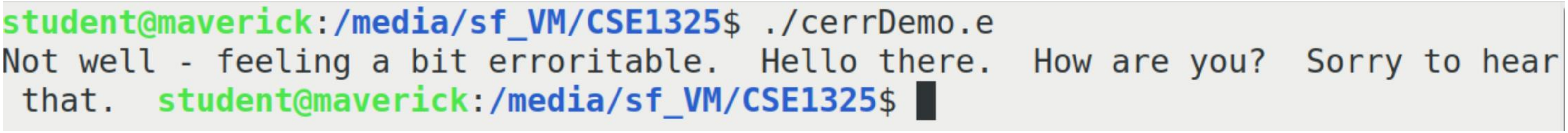
```
cout << grade;
```

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

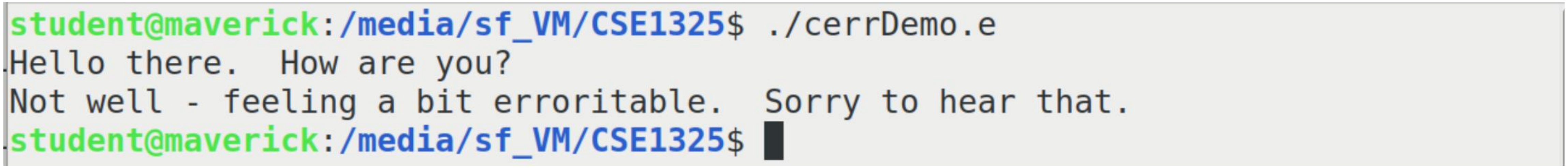
```
#include <iostream>

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

```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.
```