18 - Structs and Interoperability

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Maryville COLLEGE

Outline

Outline

Sorting

- Change into `labs/week11`

Sorting

- Change into `labs/week11`
- Open `buysell.cpp`

Sorting

- Change into `labs/week11`
- Open `buysell.cpp`
- Find the following line:

```
//sort(list.begin(), list.end());
```

Sorting

- Change into `labs/week11`
- Open `buysell.cpp`
- Find the following line:
  `//sort(list.begin(), list.end());`
- Uncomment the line:
  `sort(list.begin(), list.end());`

Sorting

- Change into `labs/week11`
- Open `buysell.cpp`
- Find the following line:
  `//sort(list.begin(), list.end());`
- Uncomment the line:
  `sort(list.begin(), list.end());`
- Now, try to compile your program.

Sorting

- Change into `labs/week11`
- Open `buysell.cpp`
- Find the following line:
  `//sort(list.begin(), list.end());`
- Uncomment the line:
  `sort(list.begin(), list.end());`
- Now, try to compile your program.
- What does the wall of text that you received tell you?

Operators and Functions

- The STL `sort` method uses the less than operator to compare objects.

Operators and Functions

- The STL `sort` method uses the less than operator to compare objects.
- `Stock` structures cannot be compared this way. Why?

Operators and Functions

- The STL `sort` method uses the less than operator to compare objects.
- `Stock` structures cannot be compared this way. Why?
- Luckily there is a way we can do this!

Operators and Functions

- The STL `sort` method uses the less than operator to compare objects.
- `Stock` structures cannot be compared this way. Why?
- Luckily there is a way we can do this!
- First, let's take a look at the general form of this operator:
  `lhs < rhs`

Operators and Functions

- The STL `sort` method uses the less than operator to compare objects.
- `Stock` structures cannot be compared this way. Why?
- Luckily there is a way we can do this!
- First, let's take a look at the general form of this operator:
  `lhs < rhs`
- We could view this as a function:
  `<(lhs, rhs)`

Overloading Operators

- C++ allows us to create **overloaded operators**.

Overloading Operators

- C++ allows us to create **overloaded operators**.
- Essentially, this lets us define operators for our custom types, like `Stock`.

Overloading Operators

- C++ allows us to create **overloaded operators**.
- Essentially, this lets us define operators for our custom types, like `Stock`.
- We can define a < operator for `Stock` like so:

```
//stock comparison
bool operator<(const Stock &lhs, const Stock &rhs)
{
    return lhs.stock_symbol < rhs.stock_symbol;
}
```

Overloading Operators

- C++ allows us to create **overloaded operators**.
- Essentially, this lets us define operators for our custom types, like `Stock`.
- We can define a < operator for `Stock` like so:

```
//stock comparison
bool operator<(const Stock &lhs, const Stock &rhs)
{
    return lhs.stock_symbol < rhs.stock_symbol;
}
```

- Be sure to add a function prototype for your operator in the proper section of your file.

```
bool operator<(const Stock &lhs, const Stock &rhs);
```

Overloading Operators

- C++ allows us to create **overloaded operators**.
- Essentially, this lets us define operators for our custom types, like `Stock`.
- We can define a < operator for `Stock` like so:

```
//stock comparison
bool operator<(const Stock &lhs, const Stock &rhs)
{
    return lhs.stock_symbol < rhs.stock_symbol;
}
```

- Be sure to add a function prototype for your operator in the proper section of your file.

```
bool operator<(const Stock &lhs, const Stock &rhs);
```

- Now, compile and test your program.

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)
- You cannot overload the following operators:

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)
- You cannot overload the following operators:
  - `::`

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)
- You cannot overload the following operators:
  - `::`
  - `.`

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)
- You cannot overload the following operators:
  - `::`
  - `.`
  - `.*`

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)
- You cannot overload the following operators:
  - `::`
  - `.`
  - `.*`
- Some operators have special rules (more on this next semester).

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)
- You cannot overload the following operators:
  - `::`
  - `.`
  - `.*`
- Some operators have special rules (more on this next semester).
- At least one parameter to the operator must be a custom made type (a `struct` or a `class`).

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)
- You cannot overload the following operators:
  - `::`
  - `.`
  - `.*`
- Some operators have special rules (more on this next semester).
- At least one parameter to the operator must be a custom made type (a `struct` or a `class`).
- Operators should be made to behave as they would by default.

Operator Overloading Rules

- The basic pattern for operator overloading is this: *return-type* `operator`*symbol*(*parameters*)
- You cannot overload the following operators:
  - `::`
  - `.`
  - `.*`
- Some operators have special rules (more on this next semester).
- At least one parameter to the operator must be a custom made type (a `struct` or a `class`).
- Operators should be made to behave as they would by default.
- The idea is to better express intent, so overloaded operators should elucidate, not obfuscate.

Selling Again

```
//sell a stock
void sell(vector<Stock> &list)
{
    //ask the user for the stock
    string stock;
    cout << "Which stock do you want to sell? ";
    cin >> stock;

    //find the stock
    auto itr = find(list.begin(), list.end(), stock);

    //if the stock is in the list, remove
    //otherwise print an error message
    if(itr != list.end()) {
        list.erase(itr);
    } else {
        cout << "Could not find stock." << endl;
    }
}
```

The Problem Here

- When we compile this, `g++` freaks out once more!

The Problem Here

- When we compile this, `g++` freaks out once more!
- Here, the trouble is this line:

```
auto itr = find(list.begin(), list.end(), stock)
```

The Problem Here

- When we compile this, `g++` freaks out once more!
- Here, the trouble is this line:

```
auto itr = find(list.begin(), list.end(), stock)
```

- This is a little bit different. What are the data types and operation in use?

The Problem Here

- When we compile this, `g++` freaks out once more!
- Here, the trouble is this line:

```
auto itr = find(list.begin(), list.end(), stock)
```

- This is a little bit different. What are the data types and operation in use?
- The following operator will help us!

```
bool operator==(const Stock &lhs, const string &rhs)
{
    return lhs.stock_symbol == rhs;
}
```

The Problem Here

- When we compile this, `g++` freaks out once more!
- Here, the trouble is this line:

```
auto itr = find(list.begin(), list.end(), stock)
```

- This is a little bit different. What are the data types and operation in use?
- The following operator will help us!

```
bool operator==(const Stock &lhs, const string &rhs)
{
    return lhs.stock_symbol == rhs;
}
```

- Note the types of the operands!

The Problem Here

- When we compile this, `g++` freaks out once more!
- Here, the trouble is this line:

```
auto itr = find(list.begin(), list.end(), stock)
```

- This is a little bit different. What are the data types and operation in use?
- The following operator will help us!

```
bool operator==(const Stock &lhs, const string &rhs)
{
    return lhs.stock_symbol == rhs;
}
```

- Note the types of the operands!
- Also, don't forget the prototype:

```
bool operator==(const Stock &lhs, const string &rhs);
```

Outline

Storing Structures in Files

- We need a file format that lets us save and restore a `struct`.

Storing Structures in Files

- We need a file format that lets us save and restore a `struct`.
- The most common way is to simply put each field on a line by itself.

Storing Structures in Files

- We need a file format that lets us save and restore a `struct`.
- The most common way is to simply put each field on a line by itself.
- We just have to make sure to read the fields in the same order we write them.

Storing Structures in Files

- We need a file format that lets us save and restore a `struct`.
- The most common way is to simply put each field on a line by itself.
- We just have to make sure to read the fields in the same order we write them.
- Will this work for `Stock` variables?

Saving Stocks

```
//save the file to disk
void save(vector<Stock> &list)
{
    //open the file
    ofstream file;
    file.open("STOCK.LST");
    if(not file) {
        //handle error
        cout << "Could not open file for writing." << endl;
        return;
    }

    //write the list to the file
    for(auto itr = list.begin(); itr != list.end(); itr++) {
        file << *itr << endl;
    }

    //close the file
    file.close();
}
```

File Stream Insertion Operator

- We need an insertion operator!

File Stream Insertion Operator

- We need an insertion operator!
- Is there anything special about insertion operators?

File Stream Insertion Operator

- We need an insertion operator!
- Is there anything special about insertion operators?
- NO!

```
ofstream& operator<<(ofstream &file, const Stock &stock)
{
    file << stock.company_name << endl
         << stock.stock_symbol << endl
         << stock.price << endl
         << stock.quantity << endl;

    return file;
}
```

File Stream Insertion Operator

- We need an insertion operator!
- Is there anything special about insertion operators?
- NO!

```
ofstream& operator<<(ofstream &file, const Stock &stock)
{
    file << stock.company_name << endl
         << stock.stock_symbol << endl
         << stock.price << endl
         << stock.quantity << endl;

    return file;
}
```

- Don't forget your prototype!

```
ofstream& operator<<(ofstream &file, const Stock &stock);
```

Loading Structs

```
//load the stocks from disk
void load(vector<Stock> &list)
{
    //open the file
    ifstream file;
    file.open("STOCK.LST");
    if(not file) {
        //return if the file does not exist
        return;
    }

    //read to the end of the file
    while(not file.eof()) {
        Stock stock;
        if(file >> stock) {
            //add all successfully read stocks
            list.push_back(stock);
        }
    }

    //close the file
    file.close();
}
```

Extraction Operator

- We need an extraction operator!

```
ifstream& operator>>(ifstream &file, Stock &stock)
{
    file >> stock.company_name
         >> stock.stock_symbol
         >> stock.price
         >> stock.quantity;

    return file;
}
```

Extraction Operator

- We need an extraction operator!

```
ifstream& operator>>(ifstream &file, Stock &stock)
{
    file >> stock.company_name
         >> stock.stock_symbol
         >> stock.price
         >> stock.quantity;

    return file;
}
```

- And its prototype:

```
ifstream& operator>>(ifstream &file, Stock &stock);
```

Extraction Operator

- We need an extraction operator!

```
ifstream& operator>>(ifstream &file, Stock &stock)
{
    file >> stock.company_name
         >> stock.stock_symbol
         >> stock.price
         >> stock.quantity;

    return file;
}
```

- And its prototype:

```
ifstream& operator>>(ifstream &file, Stock &stock);
```

- And with that, the program should be fully functional once more!