#### **Structures**

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College





#### Outline

Aggregate Data Types

Programming With Structures





#### Outline

Aggregate Data Types

Programming With Structures



 We often need to store multiple related pieces of information.



- We often need to store multiple related pieces of information.
- For instance, what does a stock holding look like?



- We often need to store multiple related pieces of information.
- For instance, what does a stock holding look like?
  - Company Name





- We often need to store multiple related pieces of information.
- For instance, what does a stock holding look like?
  - Company Name
  - Stock Symbol





- We often need to store multiple related pieces of information.
- For instance, what does a stock holding look like?
  - Company Name
  - Stock Symbol
  - Purchase Price





- We often need to store multiple related pieces of information.
- For instance, what does a stock holding look like?
  - Company Name
  - Stock Symbol
  - Purchase Price
  - Quantity





- We often need to store multiple related pieces of information.
- For instance, what does a stock holding look like?
  - Company Name
  - Stock Symbol
  - Purchase Price
  - Quantity
- How could we store this for each stock?





### One Solution: Parallel Vectors

• We could make a vector for each field:

```
vector<string> company_name;
vector<string> stock_symbol;
vector<double> price;
vector<int> quantity;
```





### One Solution: Parallel Vectors

• We could make a vector for each field:

```
vector<string> company_name;
vector<string> stock_symbol;
vector<double> price;
vector<int> quantity;
```

• This way, stock i has company\_name[i], stock\_symbol[i], price[i], and quantity[i]





### One Solution: Parallel Vectors

• We could make a vector for each field:

```
vector<string> company_name;
vector<string> stock_symbol;
vector<double> price;
vector<int> quantity;
```

- This way, stock i has company\_name[i], stock\_symbol[i], price[i], and quantity[i]
- This is, of course, error prone and awkward!





# A Better Approach

• Wouldn't it be nicer if we could do something like this?

```
Stock s;
```

Or even better, make a vector of stocks?

```
vector<Stock> portfolio;
```

It turns out, we can!





```
Stock Structure
struct Stock{
    string company_name;
    string stock_symbol;
    double price;
    int quantity;
};
```

```
Stock Structure
struct Stock{
    string company_name;
    string stock_symbol;
    double price;
    int quantity;
};
```

```
Stock Structure
struct Stock{
    string company_name;
    string stock_symbol;
    double price;
    int quantity;
};
```

• A struct is a programmer defined aggregate type.





```
Stock Structure
struct Stock{
    string company_name;
    string stock_symbol;
    double price;
    int quantity;
};
```

- A struct is a programmer defined aggregate type.
- A struct creates a custom type, which we can then use as any other variable type.





```
Stock Structure
struct Stock{
    string company_name;
    string stock_symbol;
    double price;
    int quantity;
};
```

- A struct is a programmer defined aggregate type.
- A struct creates a custom type, which we can then use as any other variable type.
- The items within a struct are called fields.





#### Outline

Aggregate Data Types

Programming With Structures





# **Declaring and Using Struct Variables**

 Structs operate like any other variable type, and fields are accessed using the '.' operator.

```
Stock s;
s.company_name = "Microsoft";
s.stock_symbol = "MSFT";
```

# **Declaring and Using Struct Variables**

 Structs operate like any other variable type, and fields are accessed using the '.' operator.

```
Stock s;
s.company_name = "Microsoft";
s.stock_symbol = "MSFT";
```

• The same is true of structs in vectors:

```
vector<Stock> list(10);
list[0].company_name = "Microsoft";
list[1].stock_symbol = "MSFT";
```





• Typically, a struct will be defined in the global scope.



- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).





- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:



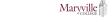


- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:
  - struct definitions.





- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:
  - struct definitions.
  - Punction Prototypes





- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:
  - struct definitions.
  - Punction Prototypes
- A typical layout for a .cpp file is:



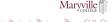


- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:
  - struct definitions.
  - Punction Prototypes
- A typical layout for a .cpp file is:
  - Includes





- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:
  - struct definitions.
  - Punction Prototypes
- A typical layout for a .cpp file is:
  - Includes
  - 2 struct definitions



- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:
  - struct definitions.
  - Punction Prototypes
- A typical layout for a .cpp file is:
  - Includes
  - 2 struct definitions
  - Function Prototypes



- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:
  - struct definitions.
  - Punction Prototypes
- A typical layout for a .cpp file is:
  - Includes
  - 2 struct definitions
  - Function Prototypes
  - Main Function





- Typically, a struct will be defined in the global scope.
- struct definitions should go before function prototypes (either in .h or .cpp files).
- A typical layout of a .h file is:
  - struct definitions.
  - Punction Prototypes
- A typical layout for a .cpp file is:
  - Includes
  - 2 struct definitions
  - Function Prototypes
  - Main Function
  - Function Definitions





Oreate the directory labs/week11.



- Oreate the directory labs/week11.
- Copy buysell.cpp from labs/week10 to labs/week11.





- Oreate the directory labs/week11.
- Copy buysell.cpp from labs/week10 to labs/week11.
- We will be modifying this file to use structures.





- Oreate the directory labs/week11.
- Copy buysell.cpp from labs/week10 to labs/week11.
- We will be modifying this file to use structures.
- Make the following changes to buysell.cpp





# buysell.cpp Includes and Structure

```
//Buy and sell stocks by symbol
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <fstream>
#include <iomanip>
using namespace std;
//type definitions
struct Stock{
    string company name;
    string stock_symbol;
    double price;
    int quantity;
```

# buysell.cpp Function Prototypes

```
//function prototypes
void buy(vector<Stock> &list);
void sell(vector<Stock> &list);
void display(vector<Stock> &list);
void load(vector<Stock> &list);
void save(vector<Stock> &list);
```





# buysell.cpp Main Function

```
int main()
    int choice;
   enum menu_choices { BUY=1, SELL, DISPLAY, QUIT };
    vector<Stock> stocks:
    //load the stocks
    load(stocks);
    //display the menu
        //get the user's choice
        cout << "
                        MENU " << endl
             << " 1.) Buv a Stock" << endl
             << " 2.) Sell a Stock" << endl
             << " 3.) Display Stocks" << endl
             << " 4.) Quit" << endl
             << end1
             << " Choice? ";
       cin >> choice:
        //do the menu choice
        if(choice == BUY) {
            buy (stocks);
        } else if(choice == SELL) {
            sell(stocks);
       } else if(choice == DISPLAY) {
            display(stocks);
        } else if(choice != QUIT) {
            cout << "Invalid selection. Please try again." << endl;
    } while (choice != OUIT);
    //save the stocks
    save(stocks);
```

# buysell.cpp Buy

```
//buv a stock
void buv (vector < Stock > & list)
    // Ask the user for a stock
    Stock stock:
    //get the stock Fields
    cout << "Company Name> ";
    cin >> stock.company_name;
    cout << "Symbol> ";
    cin >> stock.stock_symbol;
    cout << "Price> ";
    cin >> stock.price;
    cout << "Ouantity> ";
    cin >> stock.quantity;
    // add the stock to the list
    list.push_back(stock);
    //sort the stocks
    //sort(list.begin(), list.end());
```

# buysell.cpp Sell

```
//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;
* /
```

# buysell.cpp Display

```
//display our stocks
void display(vector<Stock> &list)
    //print a header
    cout << left << setw(6) << "Symbol"</pre>
         << left << setw(15) << "Company Name"
         << right << setw(10) << "Quantity"
         << right << setw(10) << "Price" << endl
         << setfill('=') << setw(41) << '=' << endl
         << setfill(' ');
    //loop over all the stocks
    for(auto itr = list.begin(); itr != list.end(); itr++) {
        cout << left << setw(6) << itr->stock_symbol
             << left << setw(15) << itr->company_name
             << right << setw(10) << itr->quantity
             << right << fixed << setprecision(2) << setw(10)
             << itr->price
             << endl;
```

# buysell.cpp Load

```
//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()) {
        string stock;
        if(file >> stock) {
            //add all successfully read stocks
            list.push back(stock);
    //close the file
    file.close();
*/
```

# buysell.cpp Save

```
//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;</pre>
        return:
    //write the list to the file
    for(auto itr = list.begin(); itr != list.end(); itr++) {
        file << *itr << endl:
    //close the file
    file.close():
    */
```