Structures

CS 211

Adam Sweeney

November 18, 2017

Wichita State University, EECS

Introduction

- Often we want to describe things in our programs that can't be simply described
- There's syntax for that

Agenda

- Chapter 10.1
- What is even a structure
- Defining and working with a structure
- Some uses and use cases

What is a Structure

What is a Structure

- Up to this point, we have only used built-in classes
  - `<string>`
  - `<iostream>`
- We'll be taking one step closer to being able to write classes of our own
- This step includes looking at a class-like data structure, a `struct`

Seriously, What is a Structure

- A structure is a way to collect many variables under a single name
- If this sounds like an array, you wouldn't be wrong
    - But that short description is where the similarities end
- The key difference is that a structure is able to hold data of different types!
- We can consider an array to be a homogeneous data structure
    - Every element of an array is of the same data type
- Conversely, a `struct` can be considered a heterogeneous data structure

You Keep Using That Word...

- Data structure?
- Won't bother with any kind of formal definition
- It is something we build for the purpose of holding data

Defining and Working With a Structure

Describing a `struct`

- We will see some familiar syntax, wrapped in some new syntax
- Let's imagine we have a book
- Books have many properties
    - Author
    - Title
    - Year
    - ISBN
- It would be tedious to declare all of these variables separately and track them individually
- We would also needlessly bloat the variable names to communicate the intent of describing a single book

Define a `struct`

```
struct Book {
    string author;
    string title;
    int year;
    string ISBN;
};
```

- The new syntax wraps around some familiar syntax
- This is how we define (not declare) a `struct`
- Why define and not declare?
  - Declare tends to connote realizing a single instance
  - We have not declared a struct called Book
  - We have created a new data type called Book using a `struct`
- Recall the general definition of a data structure
- Something we build to hold data
- Now we can *declare* as many Books as we want!

Declaring Books

```
struct Book {
    string author;
    string title;
    int year;
    string ISBN;
};

// Now to declare some Books
Book tale
Book rumpelstiltskin
```

- By defining the Book structure, we have created a new, custom data type that we can then declare like an `int`, `double`, `char`
- Each variable that the `struct` contains is called a member variable; they are members of the `struct`

Accessing the Data in a Book

- Now we've declared some books, we would naturally want to store data in them
- But how?
- Turns out we've already seen the method to do this
- We access the individual variables of a `struct` using the dot operator

```
Book tale
tale.author = "Charles Dickens";
tale.title = "A Tale of Two Cities";
tale.year = 1859;
tale.ISBN = "1542049202";
```

When We Access a Specific Member of a `struct`

- We are accessing a variable of the data type of the member
  - `tale.author` is a `string`
- This means I can pass a `struct` member variable to a function just as if it were a regular variable of that type

```
void printBasic(string auth, string title, int year);
int main()
{
    Book tale;
    // Add information to struct variable tale
    printBasic(tale.author, tale.title, tale.year);
    return 0;
}
```

Always Other Ways

- The method shown so far for defining a `struct` is the most common
- There aren't different methods, but there are permutations
- Here is a genaralized look at a `struct` definition

```
struct [TYPE NAME] {
    [MEMBER TYPE] [MEMBER NAME];
    .
    .
    .
} [VARIABLE NAME] [VARIABLE NAME] [...];
```

Explaining it

```
struct [TYPE NAME] {
    [MEMBER TYPE] [MEMBER NAME];
    .
    .
    .
} [VARIABLE NAME] [VARIABLE NAME] [...];
```

- The type name and variable names are optional, but we need to use at least one of them
- Using only the `TYPE NAME` is preferred, as it gives a name to that particular structure and allows you to declare as many variables of that type as you need
- The `VARIABLE NAME` section can come in handy if you know that for this particular program, you only need these particular `structs` declared and nothing else

Notes

- When you provide only `VARIABLE NAME`s and no `TYPE NAME`, you will unable to declare new variables of that `struct` type
- This could be seen as a pseudo-preventative measure, or you just know that you these `struct`s don't need to be more generally defined

Some Uses and Use Cases

Initializing a `struct`

- It is possible to initialize (assign at time of declaration) a `struct`

```
Book tale = {"Charles Dickens", "A Tale of Two Cities", 1859
```

- Note that the order of the data is the same as the member variables in the `struct`
- The year is not surrounded by quotes because it is not a string, but an integer
- This initialization is very similar looking to that of an array

Assigning to a `struct`

- But because a `struct` is not like an array at all, we can't use [ ] to access the members
- We always have to access each member by name
- However, it can be tedious to always have to assign values to every data member individually
- We can ease the process with a function

Initializing a struct

```
// define Book struct
Book fillBook(string auth, string title, string year, string isbn);
int main()
{
    Book tale;
    tale = fillBook("Charles Dickens", "A Tale of Two Cities",
                    1859, "1542049202");
    return 0;

}

Book fillBook(string auth, string title, string year, string isbn)
{
    Book temp;
    temp.author = auth;
    temp.title = title;
    temp.year = year;
    temp.ISBN = isbn;
    return temp;
}
```

An Explanation

- Some important notes from the `fillBook()` function
    - We can return a structure
    - This means we returned three different strings and an integer all at once
    - If a `struct` makes sense in the scheme of the variables, this is a great way to return "multiple" things from a function
    - **NOTE**: A `struct` of disjoint (unrelated) variables is bad practice
    - This makes our life a bit easier
- Now that we have this function, it will likely go with the `struct` to any other programs that may use it

Functions for `structs`

- We can have many functions to help us to things with our structures
    - A function to fill a `struct`, as shown
    - A function to fill a `struct` based on user input
    - A function to print a `struct`
    - Functions to have structs interact with other structs
- Once defined, these functions will likely travel with the `struct`

- People will tell you that you can put these functions inside your `struct`
- They're not wrong
    - But they're wrong
- In C++ we only use `structs` to hold data, not functions

Just because...

- Just because a `struct` is able to hold different data types doesn't mean that is the only way we should use them
- There are times where a `struct` may hold variables of a single data type, and it's better suited than an array
- One example is money
  - Money is better represented using two integers instead of a single double; the extra precision of a double actually gets in the way a lot of the time
  - An array of two integers would be awkward
  - A `struct` is perfectly suited to represent money

A Money `struct`

```
struct Money {
    int dollars;
    int cents;
};
```

- The `struct` alone isn't going to get us what we need, so we need to build some functions
- We'll start with two very basic functions, one to set the values, and another to print the `struct`

Setting the Values

```
Money setMoney(int d, int c)
{
    Money temp;
    temp.dollars = d;
    temp.cents = c;

    return temp;
}
```

Setting the Values

```
Money setMoney(int d, int c)
{
    Money temp;
    temp.dollars = d;
    temp.cents = c;

    return temp;
}
```

- Some notes on the above function
  - As noted prior, we can return a `struct`
  - We can **never** access the member variables of a `struct` without naming the `struct` first

Printing Money

```
void printMoney(const Money& m)
{
    cout << "$" << m.dollars << ".";
    if (m.cents < 10)
        cout << "0" << m.cents;
    else
        cout << m.cents;
}
```

Printing Money

```
void printMoney(const Money& m)
{
    cout << "$" << m.dollars << ".";
    if (m.cents < 10)
        cout << "0" << m.cents;
    else
        cout << m.cents;
}
```

- Some notes on the above function
  - Why do we need the if/else block?

Printing Money

```
void printMoney(const Money& m)
{
    cout << "$" << m.dollars << ".";
    if (m.cents < 10)
        cout << "0" << m.cents;
    else
        cout << m.cents;
}
```

- Some notes on the above function
    - Why do we need the if/else block?
        - If we had 2 dollars and 4 cents and printed the cents directly, it would have shown $2.4
        - The if/else block accounts for that and prints the leading zero when needed

Alternative Money Printing

```
void printMoney(const Money& m)
{
    cout << "$" << m.dollars << ".";
    cout.width(2);
    char prev = cout.fill('0');
    cout << m.cents;
    cout.fill(prev);
}
```

- Notes on the variation
    - Here, instead of doing a check and printing an extra character, stream manipulation is being used to ensure the leading zero gets printed
    - `setw()` could have been used to shorten the function some, but `cout.width()` was chosen because it does not require an extra library

Another Couple Notes on Printing Money

- The function does not print a new line after it prints the money
- This is because none of the regular data types do this
  - If we `cout` an integer, it doesn't automatically give us a new line
  - So why do we have that expectation when printing a `struct`?
- It is a best practice when we define our own types, to not break common expectations

- What about the parameter to the function, `const Money& m`?
  - We are combining two principles that we've already learned about
  - `const` enforces that the Money `struct` will be read-only
  - The & is the address-of operator, and it allows us to pass the Money `struct` by reference
  - This provides the same data safety as pass by value, but is more efficient because no copies are made

But, I Could Have Done This With an Array

- You're not wrong
    - But you're wrong
- While in the case of Money, an array could be used, a `struct` is still the better option from a syntactical view
- Being able to name the member variables makes the functions much easier to understand
- Storing values in an array that have different intents (one element represents dollars, but the other doesn't) breaks our expectations of an array
- Let's avoid *"If all you have is a hammer, everything looks like a nail"*

Online Example

- `https://repl.it/@sweenish/211-Ch-10-Structs`