# Structs

ITP 165 – Fall 2015

Week 9, Lecture 1

# Problem: Phone Book

- Suppose we have a phone book that supports up to 100 contacts

- For each contact, we want to store:
  - First Name (string)
  - Last Name (string)
  - Phone Number (string)

- How can we do this?

# Problem: Phone Book

- Based on what we know so far, we could make three different arrays:

```
const int MAX_CONTACTS = 100;
std::string firstNames[MAX_CONTACTS];
std::string lastNames[MAX_CONTACTS];
std::string phoneNumbers[MAX_CONTACTS];
```

# Problem: Phone Book

- Then, if we want set the info for a particular contact at index i, we could do something like:

```
firstNames[i] = "Raymond";
lastNames[i] = "Kim";
phoneNumbers[i] = "310-555-1212";
```

# Problem: Phone Book

- If we wanted to make a function that prints out the info of all the contacts, we could do something like:

```
void printContacts(std::string firsts[],
                   std::string lasts[],
                   std::string phones[])
{
    for (int i = 0; i < MAX_CONTACTS; i++)
    {
        std::cout << "First Name: " << firsts[i] << std::endl;
        std::cout << "Last Name: " << lasts[i] << std::endl;
        std::cout << "Phone: " << phones[i] << std::endl;
    }
}
```

# Problem: Phone Book

- But the annoying thing is we have three separate arrays
- Any function that wants to operate on the phone book as a whole has to accept these the arrays as parameters
- Any function that wants to operate on a single contact, has to take three parameters (one each for first, last, and phone)
- What if we later want to add in an address?

# Solution: Struct

- A ***struct*** (short for structure) allows us to group together related variables into a single variable

- So for example, we could create a struct called “Contact” that has:
  - First name
  - Last name
  - Phone number

- This way, any functions that need to operate on a contact can just be given a Contact struct, rather than all the separate variables

# Declaring a Struct, Example

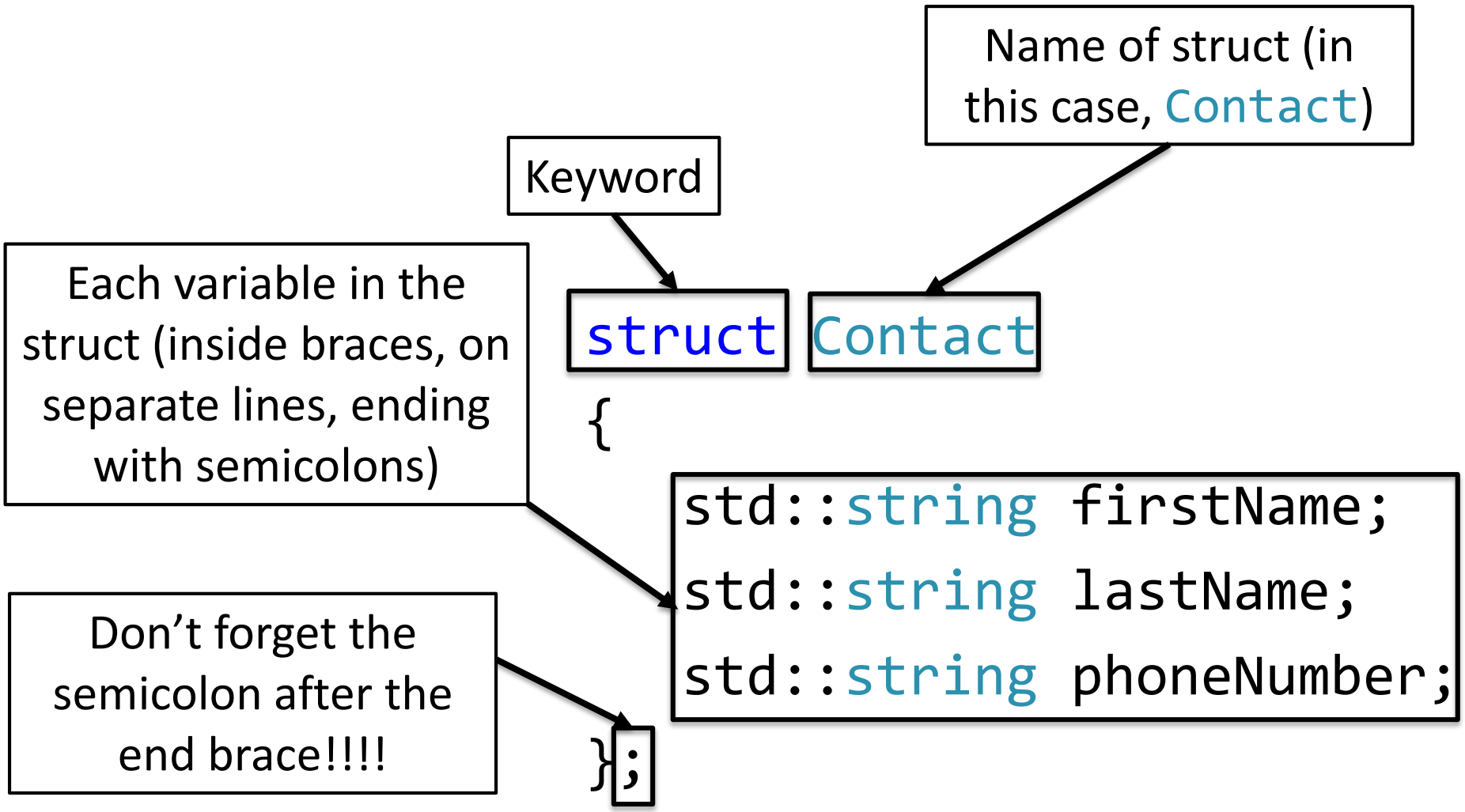
```
#include <iostream>
#include <string>

struct Contact
{
    std::string firstName;
    std::string lastName;
    std::string phoneNumber;
};

int main()
{
    return 0;
}
```

# Declaring a Struct, Example

```
#include <iostream>
#include <string>

struct Contact
{
    std::string firstName;
    std::string lastName;
    std::string phoneNumber;
};

int main()
{
    return 0;
}
```

Declaration outside of and before all functions

# Struct Declaration Syntax



# Using a struct

- Once you declare a struct, the name of the struct can now be used as if it were a type.

- So for example, given the `Contact` declaration on the previous slide…

```
// Create a Contact on the stack
Contact myContact;

// Create an array of Contacts on the stack
Contact allContacts[10];

// Dynamically allocate an array of Contacts on the heap
Contact* dynArray = new Contact[1000];
```

# Accessing Members of the Struct

- Once you create an instance of the struct, you can access the variables inside that instance (***member variables***) using the dot

- So given the `Contact` struct…

```
// Declare an instance of Contact on stack
Contact testContact;

// Set firstName member variable
testContact.firstName = "Raymond";
// Set other member variables...
testContact.lastName = "Kim";
testContact.phoneNumber = "314-159-2653";

// cout the first name
std::cout << testContact.firstName << std::endl;
```

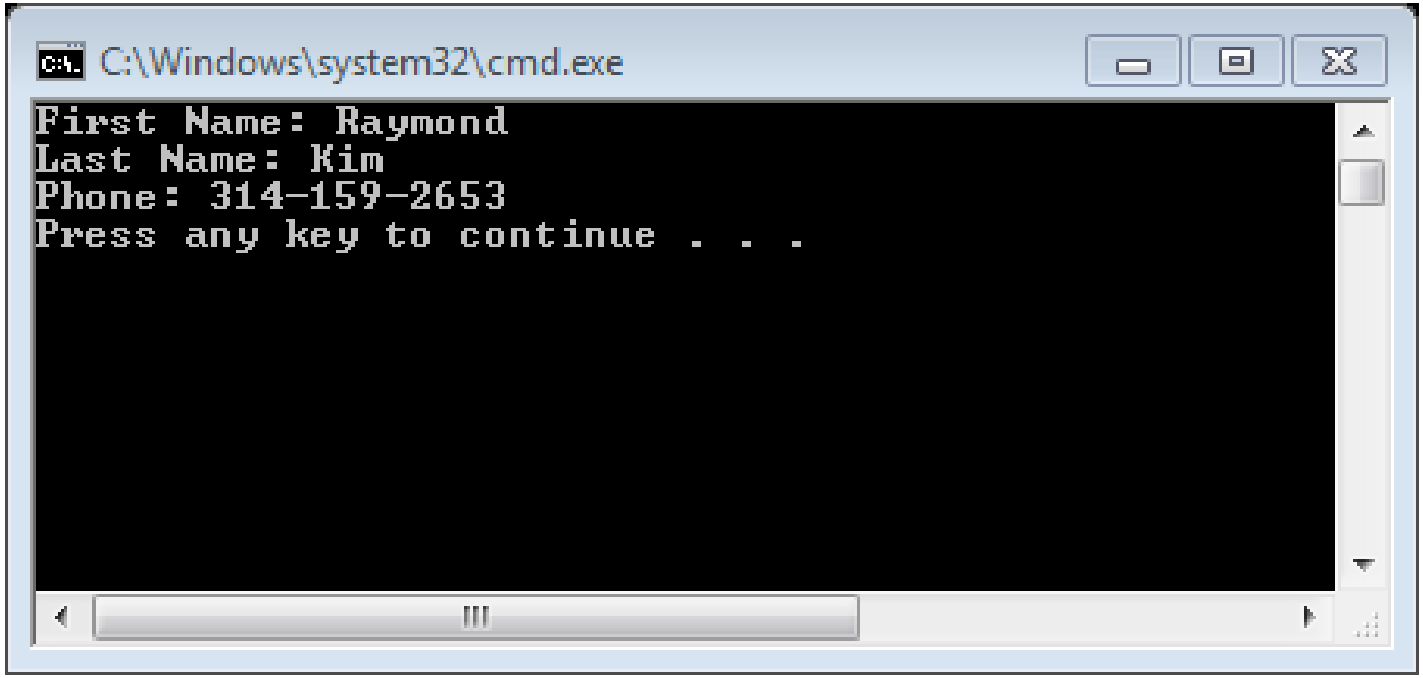
# Passing a Struct to a Function

- We can pass a struct to a function by either value or reference
- But since it's a non-basic type, we should pass it by…?
- …reference!

```
void printContact(Contact& contact) {
    std::cout << "First Name: ";
    std::cout << contact.firstName << std::endl;
    std::cout << "Last Name: ";
    std::cout << contact.lastName << std::endl;
    std::cout << "Phone: ";
    std::cout << contact.phoneNumber << std::endl;
}
```

```cpp
#include <iostream>
#include <string>

struct Contact {
    std::string firstName;
    std::string lastName;
    std::string phoneNumber;
};

void printContact(Contact& contact) {
    std::cout << "First Name: " << contact.firstName << std::endl;
    std::cout << "Last Name: " << contact.lastName << std::endl;
    std::cout << "Phone: " << contact.phoneNumber << std::endl;
}

int main() {
    // Declare an instance of Contact on stack
    Contact testContact;

    // Set member variables...
    testContact.firstName = "Raymond";
    testContact.lastName = "Kim";
    testContact.phoneNumber = "314-159-2653";

    // print it
    printContact(testContact);
    return 0;
}
```

# Example in Action

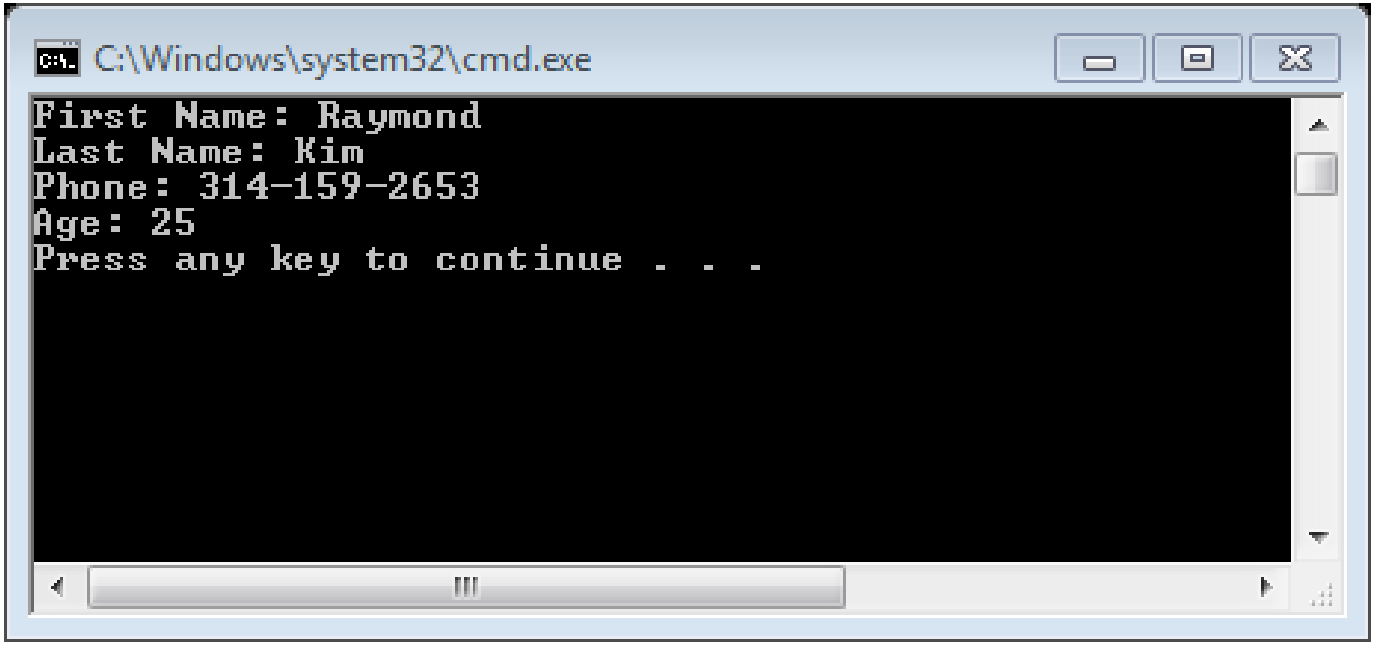


# Adding to the struct…

- Let's say we want to add a new member to the struct – an integer to represent the age
- So we need to add this field to the struct declaration
- Then just use it where it's appropriate
- No need to change function parameters!

```cpp
#include <iostream>
#include <string>

struct Contact {
    std::string firstName;
    std::string lastName;
    std::string phoneNumber;
    int age;
};

void printContact(Contact& contact) {
    std::cout << "First Name: " << contact.firstName << std::endl;
    std::cout << "Last Name: " << contact.lastName << std::endl;
    std::cout << "Phone: " << contact.phoneNumber << std::endl;
    std::cout << "Age: " << contact.age << std::endl;
}

int main() {
    // Declare an instance of Contact on stack
    Contact testContact;

    // Set member variables...
    testContact.firstName = "Raymond";
    testContact.lastName = "Kim";
    testContact.phoneNumber = "314-159-2653";
    testContact.age = 25;

    // print it
    printContact(testContact);
    return 0;
}
```

# Example w/ Age in Action



# Struct and Arrays

- If you have an array of structs, we still use square brackets and still use a dots:

```
Contact contactArray[10];

contactArray[0].firstName = "Raymond";
contactArray[0].lastName = "Kim";
contactArray[0].phoneNumber = "123-456-7890";
contactArray[0].age = 25;
```

# A Clock

- If we wanted to write a program for a basic clock, we could make a struct like this:

```
// Clock represents military time
struct Clock
{
    int hours;
    int minutes;
    int seconds;
};
```

# A Clock, Cont'd

```
// Function: resetClock
// Purpose: Resets a Clock to midnight
// Returns: Nothing
// Parameters: The Clock instance to reset
void resetClock(Clock& clock)
{
    clock.hours = 0;
    clock.minutes = 0;
    clock.seconds = 0;
}
```

# A Clock, Cont'd

```
// Function: printClock
// Purpose: Prints the current time on Clock
// Returns: Nothing
// Parameters: The Clock instance to print
void printClock(Clock& clock)
{
    std::cout << clock.hours << ":";
    std::cout << clock.minutes << ":";
    std::cout << clock.seconds << std::endl;
}
```

# A Clock, Cont'd

```
// Function: tickClock
// Purpose: Increments time on Clock by one second
// Returns: Nothing
// Parameters: The Clock instance to tick
void tickClock(Clock& clock) {
   clock.seconds++;
   if (clock.seconds == 60) {
      clock.seconds = 0;
      clock.minutes++;
      if (clock.minutes == 60) {
         clock.minutes = 0;
         clock.hours++;
         if (clock.hours == 24) {
            clock.hours = 0;
         }
      }
   }
}
```

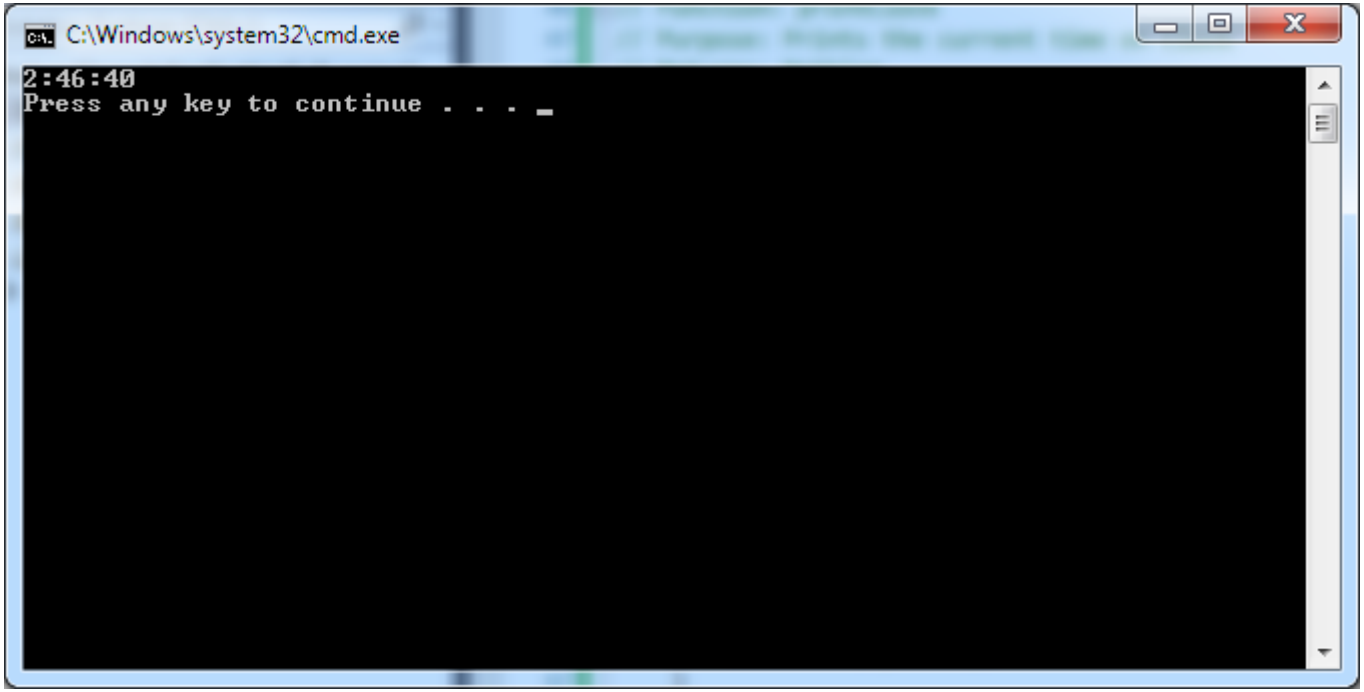
# A Clock, Cont'd

- Then I could use the struct and functions in main…

```
int main() {
   // Create a Clock instance on the stack
   Clock myClock;

   // Reset it
   resetClock(myClock);

   // Advance the clock by 10000 seconds
   for (int i = 0; i < 10000; i++) {
      tickClock(myClock);
   }

   // Print
   printClock(myClock);

   return 0;
}
```

# Clock in Action

- If I run that code, I end up with:

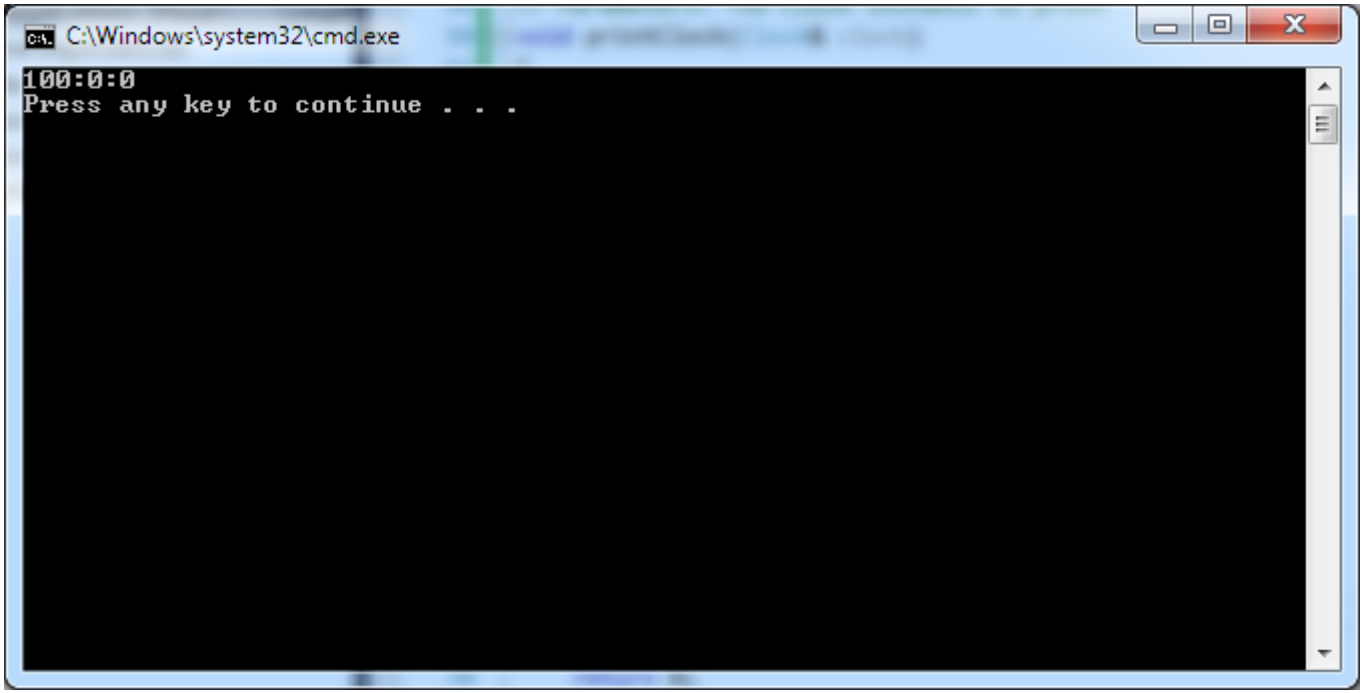


# Problem with Clock struct

- Because hours, minutes, and seconds are just properties in a struct, you could set them to invalid values…

```
int main() {
    // Create a Clock instance on the stack
    Clock myClock;

    // Reset it
    resetClock(myClock);

    // Hours can't be 100!
    myClock.hours = 100;

    // Print
    printClock(myClock);

    return 0;
}
```

# Problem with Clock struct




- Ideally, we would like to prevent code from directly changing the hours/minutes/seconds member variables
- This would ensure that the clock doesn't ever end up in an invalid state

# An annoyance with Clock struct

- Notice how we made several functions that directly operate on or access data in the `Clock`

- But every one of these functions had to take in a `Clock`, by reference, as a parameter

- And we named the functions all `somethingClock`, to signify that they operate on the `Clock`

# Lab Practical #14