COP 3331
OBJECT ORIENTED DESIGN
SPRING 2017

WEEK 3: STRUCTS, CLASSES AND OBJECTS

SCHINNEL SMALL

USF
UNIVERSITY OF SOUTH FLORIDA

RECAP: STRUCTS

ABOUT STRUCTS

- A structure or `struct` is an abstract data type that allows multiple variables to be grouped together
- It uses abstraction to capture general characteristics without details
- This data type allows
  - Values to be stored
  - Operations to be done on values

STRUCT DEFINITION

- A `struct` definition consists of:
  - The `struct` keyword, followed by a name (or tag)
    - Name typically written in title case
  - The members of the `struct`
    - May be simple or advanced data types

- General Format:

```
struct structName
{
   type1 field1;
   type2 field2;
   . . .
};
```

Example:

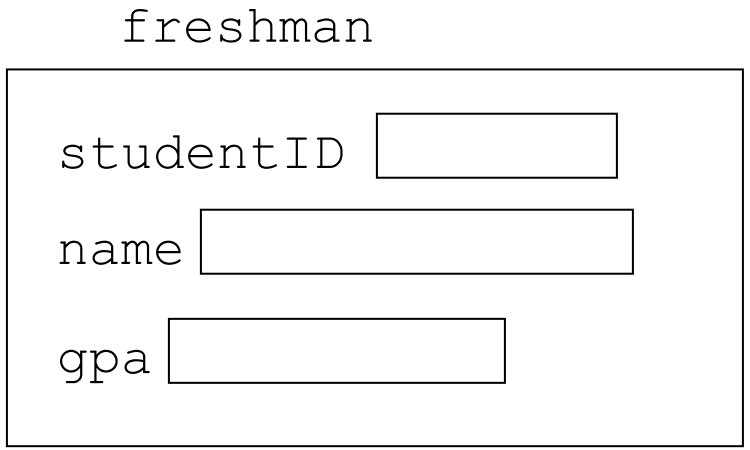
```
struct Student
{
    int studentID;
    string name;
    double gpa;
};
```

DEFINING VARIABLES

- A `struct` definition does not allocate memory

- To declare variables, the structure name must be used:

```
Student freshman;
```



ACCESSING STRUCTURE MEMBERS

- The dot operator (.) is used to refer to members of struct variables

```
cin >> freshman.studentID;
getline(cin, freshman.name);
freshman.gpa = 3.75;
```

- Members can then be used in any manner appropriate for their type
  - e.g. input, output, passed to functions etc.

STRUCT EXAMPLE

```
// This program demonstrates the use of structures.
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

struct PayRoll
{
    int empNumber;    // Employee number
    string name;      // Employee's name
    double hours;     // Hours worked
    double payRate;   // Hourly payRate
    double grossPay;  // Gross Pay
};

int main()
{
    PayRoll employee; // employee is a PayRoll structure.

    // Get the employee's number.
    cout << "Enter the employee's number: ";
    cin >> employee.empNumber;
```

```
    // Get the employee's name.
    cout << "Enter the employee's name: ";
    cin.ignore();// To skip the remaining '\n' character
    getline(cin, employee.name);

    // Get the hours worked by the employee.
    cout << "How many hours did the employee work? ";
    cin >> employee.hours;

    // Get the employee's hourly pay rate.
    cout << "What is the employee's hourly payRate? ";
    cin >> employee.payRate;

    // Calculate the employee's gross pay.
    employee.grossPay = employee.hours * employee.payRate;

    // Display the employee data.
    cout << endl << "Here is the employee's payroll data:\n";
    cout << "Name: " << employee.name << endl;
    cout << "Number: " << employee.empNumber << endl;
    cout << "hours worked: " << employee.hours << endl;
    cout << "Hourly payRate: " << employee.payRate << endl;
    cout << fixed << showpoint << setprecision(2);
    cout << "Gross Pay: $" << employee.grossPay << endl;
}
```

ABOUT STRUCTS

- Remember: You cannot display the contents of a struct by passing the entire variable to cout! Members must be passed separately:

```
cout << employee;
```
✗

```
cout << employee.name;
```
✔

- The same thing applies for comparisons:

```
if (employee1 == employee2)
```
✗

```
if (employee1.empNumber == employee2.empNumber)
```
✔

INITIALIZING A STRUCTURE

- A `struct` variable can be initialized when defined:

```
Student s = {11465, "Joan", 3.75};
```

- Or it can also be initialized member-by-member after definition:

```
s.name = "Joan";
s.gpa = 3.75;
```

INITIALIZING A STRUCTURE

- You can initialize some members:

```
Student s = {11465};
```

- But you cannot skip over members :

```
Student s = {11465 , , 3.75};
```

- And you cannot initialize in the structure definition (remember, memory isn't allocated in that step)

ARRAYS OF STRUCTS

- Structs can be defined in arrays and used in place of parallel arrays

```
const int NUM_STUDENTS = 20;
Student stuList[NUM_STUDENTS];
```

- Alternatively, you can use the array class template

```
array <Student, NUM_STUDENT> stuList;
```

- Individual Structures are accessible using subscript notations
- Members still accessible using dot notation

```
cout << stuList[5].studentID;
```

VECTORS OF STRUCTS

- To create a vector of structs:
  - with size: `vector <Student> newList (NUM_STUDENTS);`
  - without: `vector <Student> newList;`

- Once a vector of structs has values, you can use subscript notation.

- If a vector is empty (or you want to add more slots), you can use a temporary variable of type struct to fill the vector

ARRAY/VECTOR OF STRUCTS EXAMPLE

```
// This program uses an array and a vector of structures.
#include <iostream>
#include <array>
#include <iomanip>
#include <vector>
using namespace std;

struct PayInfo
{
    int hours;         // Hours Worked
    double payRate;    // Hourly Pay Rate
};

int main()
{
    const int NUM_WORKERS = 3;     // Number of workers
    array <PayInfo, NUM_WORKERS> workers;    // use array template
    int index;                     // Loop counter

    // Get employee pay data.
    cout << "Enter the hours worked by " << NUM_WORKERS
    << " employees and their hourly rates.\n";
```

ARRAY/VECTOR OF STRUCTS EXAMPLE

```
for (index = 0; index < workers.size(); index++)
    {
        // Get the hours worked by an employee.
        cout << "Hours worked by employee #" << (index + 1);
        cout << ": ";
        cin >> workers[index].hours;

        // Get the employee's hourly pay rate.
        cout << "Hourly pay rate for employee #";
        cout << (index + 1) << ": ";
        cin >> workers[index].payRate;
        cout << endl;
    }

    // Display each employee's gross pay.
    cout << "Here is the gross pay for each employee:\n";
    cout << fixed << showpoint << setprecision(2);
    for (index = 0; index < NUM_WORKERS; index++)
    {
        double gross;
        gross = workers[index].hours * workers[index].payRate;
        cout << "Employee #" << (index + 1);
        cout << ": $" << gross << endl;
    }
```

ARRAY/VECTOR OF STRUCTS EXAMPLE

```
//Create a vector for new employees
    vector <PayInfo> newWorkers;

    PayInfo temp; // create a temp structure

    //Now fill it up...
    cout << "\nHours worked by new employee: ";
    cin >> temp.hours;
    cout << "Hourly pay rate for new employee: ";
    cin >> temp.payRate;

    //... and insert it into vector
    newWorkers.push_back(temp);  // Yup, you can pass a structure
    cout << "New employee pay: $"
         << newWorkers[0].hours * newWorkers[0].payRate
         << endl;
}
```

ABOUT STRUCTS

- Note: Arrays *of* structures is different than arrays *in* structures
- An array of structures is an array of type `struct`
- An array in a structure is an array that is a member of a `struct`:

```
struct Example
{
   int list [10];
   double values;
};
```

- To access values in a member that is an array, the [ ] notation applies to the member:

```
Example var;
var.list [3] = 5;
```

NESTED STRUCTS

- You are also allowed to create nested structures (i.e. `structs` in `structs`)

```
struct Costs
{
   double wholesale;
   double retail;
};
```

```
struct Item
{
   string partNum;
   Costs pricing;
};
```

- For each nested structure, you would need an additional dot operator to access the member

```
Item widget;
widget.pricing.wholesale = 100.0;
```

- You need to get to the primitive member for the value to be stored!

STRUCTS AND FUNCTIONS

- You can pass an entire `struct` variable to a function (as seen in the previous example).
  - For your own functions, you would declare a parameter of type struct
- You can also pass members of a `struct` (as they are technically simple variables
- You can return a `struct` from a function
  - This allows you to bypass that ‘single value’ limitation for return statements

STRUCTS AND POINTERS

- A pointer can be of type struct
  - used for dynamic memory allocation

```
Student *stuptr = nullptr;
```

  - (nullptr was introduced in C++11)

- To access members of this pointer, use the member access arrow ->

```
stuptr = new Student;
stuptr -> studentID = 123456;
```

WHEN TO USE -> OR .

- `s -> m;` s is a structure pointer and m is a member
  - equivalent to `(*s).m;`
- `*s.m;` s is a structure variable and m is the pointer
- *s->m; s is a structure pointer and m is a pointer member
  - -> dereferences s; * dereferences m
  - equivalent to `*(*s).m;`

CLASSES AND OBJECTS

CLASSES

- Recall: A class is like a blue print or concept

- Objects can be created (instantiated) from classes
  - An object like is a *realization* of the concept
  - Objects contain data (variables) and attributes (functions/methods) based on the class definition

- Objects tend to enforce *encapsulation*, which allows the object to act as a self-governing entity

CLASSES

- A class is very similar to a struct with one notable difference:
  - The members of a struct default to *public* access (and can be accessed directly in code) when no access specifiers are used…
  - Whereas the members of a class default to *private* access (and cannot be accessed directly) when no access specifiers are used
    - private access implement the data hiding feature of objects

CLASS DEFINITION

- To define a class, use the following syntax

```
class className
{
    private:
        type1 field1;
        . . .
    public:
        type2 field2;
        . . .
};
```

- A class definition typically has both private and public members
  - private members are typically variables
  - public members are typically functions

CLASS DEFINITION

- There is no restriction on the order in which public or private members can be placed
  - You can declare public members before private or vice versa
- There is also no rule that sates that public and private members must be grouped together
  - If they are not, each member will need its own access specifier
- Tip: pick a style and be consistent

WHERE TO PLACE A CLASS DEFINITION

- As class definitions can get long/complex, programmers tend to define class in their own files
  - The files are considered to be user-created header files (.h extension), and are stored in the same directory as the .cpp file that contains the main function
  - The are called *class specification files*
  - We include it like a standard header file, except we use “ ” to indicate that the file is within the same directory

- Important: classes *cannot* execute by themselves!
  - The .cpp file that contains the main function becomes the *driver program* for the class

TYPES OF MEMBER FUNCTIONS

- The member functions of a class may include:
  - Accessor functions – functions that retrieves a value from a class member's variable
    - They typically use the word "get" in the function name
    - They are also called "getter functions"
    - They may include the keyword const which ensures that the function cannot modify the object's contents
  - Mutator Functions – functions that can change the value in a class member's variable
    - They typically use the word "set" in the function name
    - They are also called "setter functions"

TYPES OF MEMBER FUNCTIONS

- The member functions of a class may include:
  - Constructors – functions that are automatically called when a class object is created
    - Share the same name as the name of the class
    - Have no return type (not even void)
    - May be overloaded (depending on object initialization requirements)
    - C++ provides a default constructor (with no parameters) if one is not explicitly provided, but there can only be one!
      - Note: a constructor with default parameters is considered a default constructor
    - as of C++ 11, the default constructor is not created if you created your own

TYPES OF MEMBER FUNCTIONS

- Types of member functions may include:
  - Destructors – functions that are automatically called when an object is destroyed
    - Share the same name as the name of the class preceded by the tilde character (~)
    - Have no return type
    - Cannot have parameters
    - Cannot be overloaded
    - May not be explicitly created as newer features can perform some of a destructor's features automatically

MEMBER FUNCTION LOCATIONS

- Member functions may be defined outside or inside of the class definition

- If defined outside the class definition:
  - the function prototype is listed in the class definition
  - the scope resolution operator (::) must be used to associate the function as a member of the class
  - Member functions declared outside the class are typically stored in their own .cpp files called class *implementation files*

- If defined inside the class definition
  - the function is declared *inline* so no :: is needed
  - great strategy for smaller function definitions

CLASS EXAMPLES

CLASS EXAMPLE – HEADER WITH MEMBER FUNCTIONS OUTSIDE CLASS DEFINITION

```
#include <string>
using namespace std;

class Account
{
private:
    string name;

public:
    void setName(string accountName);
    string getName() const;

}; // end class Account definition. Don't forget the semicolon!

// mutator function that sets the account name in the object
void Account::setName(string accountName)
    {
        name = accountName;
    }

// accessor function that retrieves the account name from the object
string Account::getName() const
{
    return name;
}
```

CLASS EXAMPLE – HEADER WITH INLINE FUNCTIONS

```
#include <string> // enable this program to use C++ string data type
using namespace std;

class Account
{
private:
    string name; // data member containing account holder's name

public:
    // mutator function
    void setName(string accountName)
    {
        name = accountName; // store the account name
    }

    // accessor function
    string getName() const
    {
        return name; // return name's value to this function's caller
    }
}; // end class Account. Don't forget the semicolon!
```

DRIVER FOR CLASS WITH INLINE FUNCTIONS

```
#include <iostream>
#include <string>
#include "Account.h"    // need to include the file with the class definition
using namespace std;
int main()
{
    Account myAccount; // create Account object myAccount

    // show that the initial value of myAccount's name is the empty string
    cout << "Initial account name is: " << myAccount.getName();

    // prompt for and read name
    cout << "\nPlease enter the account name: ";

    string theName;
    getline(cin, theName); // read a line of text

    myAccount.setName(theName); // put theName in myAccount

    // display the name stored in object myAccount
    cout << "Name in object myAccount is: "
    << myAccount.getName() << endl;
}
```

Sample Output

```
Initial account name is:
Please enter the account name: John Doe
Name in object myAccount is: John Doe
Program ended with exit code: 0
```

CONSTRUCTOR EXAMPLE (OUTSIDE CLASS DEFINITION)

```
#include <string>
using namespace std;

class Account
{
private:
    string name;

public:
    Account(string n);  // constructor prototype
    void setName(string accountName);
    string getName() const;
};

// This constructor will initialize the account name.
Account::Account(string n)
{
    name = n;
}

// mutator function
void Account::setName(string accountName)
{    name = accountName;  }

// accessor function
string Account::getName() const
{    return name;   }
```

With this constructor, the only line the changes in the driver program is the object declaration line:

```
Account myAccount{"Johnny Appleseed"};
```

Sample Output

```
Initial account name is: Johnny Appleseed
Please enter the account name: Betty Cashflow
Name in object myAccount is: Betty Cashflow
Program ended with exit code: 0
```

CONSTRUCTOR EXAMPLE INSIDE CLASS DEFINITION

- Standard syntax:

```
Account(string n)
 {
      name = n;
 }
```

With explicit keyword:

```
explicit Account (string n):
name {n}
{
    // Empty body
}
```

- The explicit keyword can be used to prevent the argument of an constructor from being converted to an object of the class
  - This is implicit conversion
  - Applies to constructors defined with one parameter

CLASS EXAMPLE WITH MORE SPECIFIC FUNCTIONS

```
#include <string>
using namespace std;

class Account
{
private:
    string name;
    int balance{0};  // data member with default initial value

public:
    // constructor
    Account(string n, int initBal) : name{n}
    {
        // could have written name = n here instead of appending it
        // to the function header

        // validate that the initBal is greater than 0; if not,
        // data member balance keeps its default initial value of 0
        if (initBal > 0)
        {                       // if the initBal is valid
            balance = initBal; // assign it to data member balance
        }

    }
```

CLASS EXAMPLE WITH MORE SPECIFIC FUNCTIONS

```
    // function that deposits (adds) only a valid amount to the balance
    void deposit(int depositAmount)
    {
        if (depositAmount > 0)  // if the depositAmount is valid
        {
            balance = balance + depositAmount; // add it to the balance
        }
    }

    // function returns the account balance
    int getBalance() const
    {   return balance;     }

    // mutator
    void setName(string accountName)
    {   name = accountName;     }

    // accessor
    string getName() const
    {   return name;    }

};
```

DRIVER PROGRAM FOR PREVIOUS EXAMPLE

```
#include <iostream>
#include "Account.h"
using namespace std;

int main()
{
    Account account1{"Jane Green", 50};
    Account account2{"John Blue", -7};

    // display initial balance of each object
    cout << "account1: " << account1.getName() << " balance is $"
    << account1.getBalance();
    cout << "\naccount2: " << account2.getName() << " balance is $"
    << account2.getBalance();

    cout << "\n\nEnter deposit amount for account1: "; // prompt

    int depositAmount;
    cin >> depositAmount; // obtain user input
    cout << "adding " << depositAmount << " to account1 balance";
    account1.deposit(depositAmount); // add to account1's balance
```

DRIVER PROGRAM FOR PREVIOUS EXAMPLE

```
    // display balances
    cout << "\n\naccount1: " << account1.getName() << " balance is $"
    << account1.getBalance();
    cout << "\naccount2: " << account2.getName() << " balance is $"
    << account2.getBalance();

    cout << "\n\nEnter deposit amount for account2: "; // prompt
    cin >> depositAmount; // obtain user input
    cout << "adding " << depositAmount << " to account2 balance";
    account2.deposit(depositAmount); // add to account2 balance

    // display balances
    cout << "\n\naccount1: " << account1.getName() << " balance is $"
    << account1.getBalance();
    cout << "\naccount2: " << account2.getName() << " balance is $"
    << account2.getBalance() << endl;
}
```

ANNOUNCEMENTS

ANNOUNCEMENTS

- Assignment 2 on canvas
  - Based on arrays, structs and classes
  - Due Thursday February 2nd

- Check out the Programming Resource Center (PRC) for additional assistance with code
  - Check it out here: http://www.usf.edu/engineering/cse/undergraduate/prc-info.aspx