Foundation of Computer Science: Struct

Dr Kafi Rahman

Assistant Professor

Truman State University

Abstract Data Type

- Grouping a set of data to describe a person or a thing.
- This grouping is called Abstract Data Type (ADT). For example,
  - movie
  - car
  - person

struct
<CAR>

Abstract Data Type

- This grouping of data to describe a person or a thing is called data abstraction.
  - an ADT is defined by the programmer
  - it has one or more data fields which may be primitive data types

Abstract Data Type (cont)

- For example, to describe a clock we can group three integer data.
- Then the data fields of the Clock ADT might be
  - hours, a field that can take on values from 1 to 12 inclusive
  - minutes, a field that can take on values from 0 to 59 inclusive
  - seconds, a field that can take on values from 0 to 59 inclusive
- and the operations might involve adding and subtracting time values with correct carries, and comparing them

C++ Structures

- The primary C++ mechanism for building ADTs is the struct
- imagine you wish to build a system for maintaining information about movies
- you might define a Movie structure like this:

```
1  struct Movie
2  {
3    string title;
4    string director;
5    unsigned year_released;
6    double running_time;
7  };
```

C++ Structures (cont)

- the data fields are attributes of the structure
- the closing curly brace is followed by a semicolon

```
1  struct Movie
2  {
3    string title;
4    string director;
5    unsigned year_released;
6    double running_time;
7  };
```

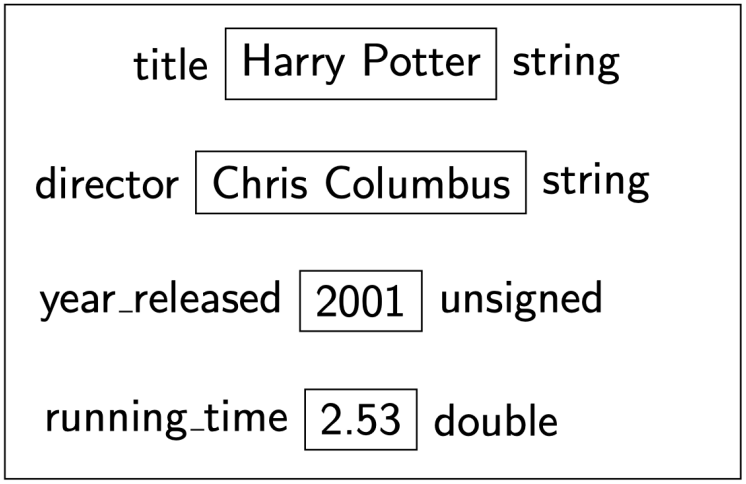
A Structure Variable

- a struct is a template or a blueprint
- a struct is an ADT.
- we can declare a variable of this type, using the structure name

```
Movie movFan {"Harry Potter", "Chris
Columbus", 2001, 2.53};
```

A Structure Variable (cont)

- movie is a variable that has four attributes or data fields.
- we diagram this variable schematically like this:




```
Movie movFan {"Harry
Potter", "Chris
Columbus", 2001, 2.53};
```

Accessing Structure Members (cont)

- to access a individual structure member, we use the dot operator and dot notation

```
cout << movie.title << endl; // Harry Potter;
```

Using Structure: Circle Example

```
1  // This program stores data
2  // about a circle in a structure.
3  #include <iostream>
4  #include <cmath>
5  // For the pow function
6  #include <iomanip>
7  using namespace std;
8
9  // Constant for pi.
10
11 const double PI = 3.14159;
12
13 // Structure declaration
14 struct Circle {
15   double radius; // A circle's radius
16   double diameter; // A circle's diameter
17   double area; // A circle's area
18 };
```

Using Structure: Circle Example

```
22  int main()
23  {
24    // Define a structure variable
25    Circle c;
26    // Get the circle's diameter.
27    cout << "Enter the diameter of a circle: ";
28    cin >> c.diameter;
29    // Calculate the circle's radius.
30    c.radius = c.diameter / 2;
31
32    // Calculate the circle's area.
33    c.area = PI * pow(c.radius, 2.0);
34    // Display the circle data.
35    cout << fixed << showpoint << setprecision(2);
36    cout << "The radius and area of the circle are:\n";
37    cout << "Radius: " << c.radius << endl << "Area: " << c.area << endl;
38  }
```

Using Structure: Circle Example

Program Output with Example Input Shown in Bold

```
Enter the diameter of a circle: 10 [Enter]
The radius and area of the circle are:
Radius: 5
Area: 78.54
```

Using Structure: PayRoll Example

```
1  // This program demonstrates
2  // the use of structures.
3  #include <iostream>
4  #include <string>
5  #include <iomanip>
6  using namespace std;
7
8  struct PayRoll
9  {
10   int empNumber;
11   string name;
12   double hours;
13   double payRate;
14   double grossPay;
15 };
```

Using Structure: PayRoll Example

```
18  int main() {
19    PayRoll employee; // employee is a PayRoll structure.
20    // Get the employee's number.
21    cout << "Enter the employee's number: ";
22    cin >> employee.empNumber;
23    // Get the employee's name.
24    cout << "Enter the employee's name: ";
25    cin.ignore(); // To skip the remaining ‘\n' character
26    getline(cin, employee.name);
27    // Get the hours worked by the employee.
28    cout << "How many hours did the employee work? ";
29    cin >> employee.hours;
30    // Get the employee's hourly pay rate.
31    cout << "What is the employee's hourly payRate? ";
32    cin >> employee.payRate;
```

Using Structure: PayRoll Example

```
34    // Calculate the employee's gross pay.
35    employee.grossPay = employee.hours * employee.payRate;
36
37    // Display the employee data.
38
39    cout << "Here is the employee's payroll data:\n";
40    cout << "Name: " << employee.name << endl;
41    cout << "Number: " << employee.empNumber << endl;
42    cout << "Hours worked: " << employee.hours << endl;
43    cout << "Hourly payRate: " << employee.payRate << endl;
44    cout << fixed << showpoint << setprecision(2);
45    cout << "Gross Pay: $" << employee.grossPay << endl;
46
```

Using Structure: PayRoll Example

Program 11-1 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter the employee's number: 489 [Enter]
Enter the employee's name: Jill Smith [Enter]
How many hours did the employee work? 40 [Enter]
What is the employee's hourly pay rate? 20 [Enter]
Here is the employee's payroll data:
Name: Jill Smith
Number: 489
Hours worked: 40
Hourly pay rate: 20
Gross pay: $800.00
```

Using Structure: PayRoll Example

- The contents of a structure variable cannot be displayed by passing the entire variable to cout. For example, assuming employee is a PayRoll structure variable, the following statement will not work:

```
cout << employee << endl;
```

- // Will not work!

- Instead, each member must be separately passed to cout.

Initializing a Struct Variable

- a struct variable can be initialized when it is declared by filling all the fields in order (note no assignment operator)

```
Movie movFan {"Harry Potter", "Chris
Columbus", 2001, 2.53};
```

Initializing a Struct Variable (cont)

- or by assigning them one-by-one after declaration:

```
Movie movie;
movie.director = "Chris Columbus";
movie.year_released = 2001;
movie.title = "Harry Potter";
movie.running_time = 2.53;
```

Vectors of Structs

- it is perfectly legal to have a single struct variable
- but the real power of structs comes with collections of structs i.e., arrays or vectors of structs
- a vector of structs is created just like any vector
- first, we need the struct ADT definition

```
1  struct Movie
2  {
3    string title;
4    string director;
5    unsigned year_released;
6    double running_time;
7  };
```

Vectors of Structs (cont)

- then we use it to create vectors

```
1  int main()
2  { // declare a vector variable that can store Movie variables
3    vector<Movie> movies;
4
5    movies.push_back({"Psycho", "Hitchcock", 1960, 1.82});
6    movies.push_back({"Vertigo", "Hitchcock", 1958, 2.13});
7    movies.push_back({"Repulsion", "Polanski", 1965, 1.75});
8
9    for (Movie movie : movies)
10   {
11     cout << movie.title << ' ' << movie.year_released << endl;
12   }
13   return 0;
14 }
```

Passing Structs to Functions

- a struct variable may be passed to a function like other primitive variables (int, float, etc).

```
void displayMovie(Movie m)
{   cout<<"\nMovie title: "<<m.title
        <<"\nDirector: "<<m.director
        <<"\nYear: "<<m.year_released
        <<"\nTime: "<<m.run_time;
}

// driver program
int main()
{   Movie movFan {"Harry Potter",
          "Chris Columbus", 2001, 2.53};
    // passing the movie variable
    displayMovie(movFan);
    return 0;
}
```

Pass by Reference

- almost always used in real programs
- pass by reference if changes are needed in the calling scope

```
void displayMovie(Movie & m)
{   cout<<"\nMovie title: "<<m.title
        <<"\nDirector: "<<m.director
        <<"\nYear: "<<m.year_released
        <<"\nTime: "<<m.run_time;
}

// driver program
int main()
{   Movie movFan {"Harry Potter",
          "Chris Columbus", 2001, 2.53};
    // passing the movie variable
    displayMovie(movFan);
    return 0;
}
```

Pass by Constant Reference

- almost always used in real programs
- pass by constant reference if no changes are allowed

```
void displayMovie(const Movie & m)
{   cout<<"\nMovie title: "<<m.title
        <<"\nDirector: "<<m.director
        <<"\nYear: "<<m.year_released
        <<"\nTime: "<<m.run_time;
}

// driver program
int main()
{   Movie movFan {"Harry Potter",
          "Chris Columbus", 2001, 2.53};
    // passing the movie variable
    displayMovie(movFan);
    return 0;
}
```

Copying Struct Variables

- unlike arrays, it is perfectly legal to copy one struct to another in a single statement

```
Movie movie_x {"Psycho", "Hitchcock", 1960,
1.82};
Movie movie_y;
movie_y = movie_x;
```

- now movie_y is an exact duplicate of movie_x
  - copied member by member

Comparing Struct Variables

- like arrays, you cannot compare struct variables with relational operators (relops)
- what would this even mean in our program
  - movie_x < movie_y
    - do we mean alphabetic less-than by title?
    - numeric less-than by release date?

- rather, we should to write a function to compare two struct variables

Comparing Struct Variables

- write a function for alphabetic less-than by title, with ties broken by date

```
7  // compare by title of the movie
8  bool less_than_alphabetic(const Movie & m_x, const Movie & m_y)
9  { if (m_x.title == m_y.title)
10   { return m_x.year_released < m_y.year_released;
11   }
12   return m_x.title < m_y.title;
13 }
```

Printing Struct Variables

- we cannot output a struct variable like this: cout << movie_x;
- we have to output individual members

```
14  int main()
15  {
16    string title = "Harry Potter";
17    string director_name = "Director";
18    unsigned release_date = 2001;
19    float running_time = 1.20;
20
21    unsigned hours = static_cast<unsigned> (running_time);
22    unsigned minutes = static_cast<unsigned> ((running_time-hours) * 60.0);
23
24    string output = title + "; " + director_name;
25    output += " (" + to_string(release_date) +") ";
26    output += to_string(hours) + " hr " + to_string(minutes) + " min";
27
28    cout<<output;
29  }
30
31  Display: Harry Potter; Director (2001) 1 hr 12 min
```

Printing Struct Variables

- we cannot output a struct variable like this: cout << movie_x;
- we could write a function print movie, but it is much better conceptually to write this function:

```
1  string to_string(const Movie& m)
2  {
3    string result = m.title + "; " + m.director +
4      " (" + to_string(m.year_released) + ") ";
5
6    unsigned hours = static_cast<unsigned>(m.running_time);
7    unsigned minutes = static_cast<unsigned>((m.running_time - hours) * 60.0);
8
9    result += to_string(hours) + " hr " + to_string(minutes) + " min";
10   return result;
11 }
12
13 Output: Psycho; Hitchcock (1960) 1 hr 49 min
```

Nested Structures

- a member can be of any data type
- including a programmer-defined struct ADT

```
1  // Defining a Time Structure
2  struct Time
3  { unsigned hour;
4    unsigned minute;
5  };
6
7  // Movie structure uses Time ADT
8  struct Movie
9  { string title;
10   string director;
11   unsigned year_released;
12   Time running_time; //ADT
13 }
```

Nested Structures

- nested members can be initialized via nested initializer lists
- nested members are accessed via nested dots

```
1  Movie movie_x {"Psycho", "Hitchcock", 1960, {1, 49}};
2
3  Movie movie_y;
4  movie_y.title = "Vertigo";
5  movie_y.running_time.hour = 2;
6  movie_y.running_time.minute = 8;
```

Nested Structures

- the to string function now becomes:

```
1  // display the struct as string
2  string to_string(const Movie & m)
3  { string result = m.title + "; " + m.director +
4        " (" + to_string(m.year_released) + ") " +
5        to_string(m.running_time.hour) + " hr " +
6        to_string(m.running_time.minute) + " min";
7
8    return result;
9  }
```

Pointers to Structure Variables

- a pointer variable can point to a structure location in memory

```
Movie movie {"Psycho", "Hitchcock", 1960, {1, 49}};
Movie * mptr = &movie;
```

- immediately there is a problem, however to access a member, we use the dot operator after dereferencing the pointer:
  - but this doesn't work, because the precedence of dot is higher than the precedence of dereference

```
cout << *mptr.title;
```

Pointers to Structure Variables

- instead we have to do this:

```
Movie movie {"Psycho", "Hitchcock", 1960, {1, 49}};
Movie* mptr = &movie;
cout << (*mptr).title;
```

- this syntax is required in C, and works in C++, but is considered awkward

- instead C++ uses the dereference-then-select operator `->`

```
cout << mptr->title;
```

- this operator means: first dereference mptr, then go to the title field of the thing mptr is pointing to, and print that

Dynamically Allocating Structures

- with the ability to have pointers to structure variables, we can dynamically allocate them
- this is essential in C, rarely done in C++ until CS310

```
1  Movie* mptr = new Movie; // allocating memory
2  // assigning values to this struct
3  mptr->title = "Billy Jack";
4  mptr->director = "Tom Laughlin";
5  mptr->year_released = 1971;
6  mptr->running_time.hour = 1;
7  mptr->running_time.minute = 54;
8
9  cout << to_string(*mptr) << endl; delete mptr;
10
11
12 Output:
13 Billy Jack; Tom Laughlin (1971) 1 hr 54 min
```

Not Using

- we will not use section 11.10 and 11.11

Reading: Chapter 11

- Read Sections
  - all, except 11.11
- Try Example Programs
  - all, except 11-9, 11-10, 11-11, 11-12
- Checkpoint Exercises
  - 11.1, 11.2, 11.3, 11.4, 11.5, 11.7, 11.8, 11.10, 11.11, 11.13, 11.14, 11.15
- Review Questions
  - Short Answer
    - 1, 3, 4, 6, 8
  - Fill in the Blanks
    - 11, 13, 15, 16
  - Algorithm Workbench
    - 17, 19, 20
  - True False
    - 30, 31, 32, 33, 34, 37, 39, 40, 41, 42
  - Find Errors
    - 45, 46, 47, 48, 49, 50, 52

Reading: Chapter 11

- Programming Challenges
  - 1, 4, 6, 7, 8, 11, 13
    - Make assumptions if something is not clear from the problem description

- Assignment
  - Complete Checkpoint Exercises
  - Complete Programming Challenges
    - create a zip archive and submit them on the Blackboard by Nov 30