structs

- Arrays are useful for storing a collection of data elements of the **same** data type
- What about storing a collection of data elements of **different** data types?
- Related information can be placed in a ***structure***, which has a general format as follows:

```
struct StructName
{
   // variable declarations
};
```

struct Definition

- structs store a collection of data elements of ***different*** data types
- For example, what if we wanted to keep the following information on a particular employee:
  - employee id
  - SS#
  - number of children
  - salary
  - citizen
- The elements have different data types, so we can't conveniently use an array. Instead we will use a struct

Structure Declaration

To store this information:

- employee id
- SS#
- number of children
- salary
- citizen

We would begin by defining a structure :

```
struct Employ
{
    int id;
    int ssnum;
    int numchild;
    double salary;
    bool bCitizen;
};
```

Struct Terminology

For this struct:

```
struct Employ
{
   int id;
   int ssnum;
    int numchild;
   double salary;
   bool bCitizen;
};
```

- `Employ` is the **identifier name** and a new data type.
- The individual components id, ssnum, etc. are called **members.**

Notes on Structures

- A semicolon is required after the closing brace of the structure declaration
- The structure declaration does not create a variable
- It just tells the compiler what that structure is made of
- The struct declaration is usually placed above the main

Variable Declaration

- As with all data types, in order to use our new data type **`Employ`** we must *allocate* storage space by *declaring* variables of this data type:

```
Employ sEngineer, sTech;
```

- This will allocate space for two variables called **`sEngineer`** and **`sTech`**, each containing the previously described members **`id`**, **`ssnum`**, etc.
- Each of these variables is a separate instance of **`Employ`**

Dot Operator

- To access a struct member, we use the ***dot operator*** (period between struct variable name and member name).
- In the variable **`sEngineer`** of data type **`Employ`** we can make the assignments:

```
sEngineer.id = 12345;

sEngineer.ssnum = 534334343;

sEngineer.numchild = 2;

sEngineer.salary = 45443.34;

sEngineer.bCitizen = true;
```

Practice

- Read Pacific Soccer scores from a file. Calculate the Pacific team's record.
- How long is their longest winning streak?

```
Pacific 5 NorthwestChristian 0
Redlands 2 Pacific 1
LaVerne 0 Pacific 6
.....
Pacific 1 PacificLutheran 0
```

The home team is listed first.

No team name contains a space.

Build a struct

Read the data from the file

Notes on Structures

- You cannot output the entire contents of a struct variable by simply using its name
  - `cout << sEngineer;  // ERROR!`
- Similarly, you cannot compare two struct variables by using their name
  - `if(sEngineer == sTech)  // ERROR!`

struct Definition

- structs are ***user defined data types*** that can be used to declare variables. The variables that appear inside of the struct definition are ***members*** of the structure.

Payroll Problem

- Consider the following structure:

```
struct PayRoll
{
   int     employeeNumber;
   string  name;
   double  hoursWorked,
           payRate,
           grossPay;
};
```

Payroll Problem

- Declare a **`PayRoll`** variable **`deptHead`** and assign the **`employeeNumber`**, **`name`**, and **`payRate`** with the values **`123`**, **`Joe Smith`**, and **`10.00`**.

Time Problem

- Consider the following struct:

```
struct Time
{
   int hours,
       minutes,
       seconds;
};
```

- Write the C++ code that will read in a military time in the form hh:mm:ss and place hh into hours, mm into minutes, and ss into seconds. Error check to make sure that hh is in the range of 0-23, mm is in the range of 0-59, and ss is in the range of 0-59.

Displaying/Comparing structs

- Which of the following C++ statements are legal given variables time1 and time2 of type Time exist?

a)
```
cout << time1 << time2;
```

b)
```
if(time1 == time2)
   {
     cout << "times are equal";
   }
```

c)
```
cout << time1.hours;
```

d)
```
cin >> time1;
```

e)
```
cin >> time1.Hours;
```

Initializing Structs UPDATED

- Use an initializer list
  - **`Employ manager = {12345, 534334356, 1, 76899, true};`**
- You can initialize only some of the members in a struct, but members that follow a non initialized member must also be not initialized
  - **`Employ manager = {12345,534334356,1};`**
  - **`Employ manager = { 12345,,,, true};`**

Initializing Structs

- You cannot initialize structures in the declaration

```
struct Employ
{
  int id = 12345;
  int ssnum = 534334356;
  int numchild = 1;
  float salary = 75000;
  bool bCitizen = true;
};
```

ERROR!

- Why?

Passing structs to Functions

- structs can be passed to functions by reference or value in the same manner that other data types have been passed
- Generally, passing structs by reference is preferred since passing by value requires a local copy of the struct to be created within the function's variables

Example

```
struct Date
{
   int day,
       month,
       year;
};
```

- Create a date variable equal to Monday, November 22, 2010
- Write a function that accepts a Date and prints the date out in the form day-month-year

Arrays of structs

- It is possible to declare an array of structs
- A datafile called athletes.txt exists which contains an unknown amount of information where each line of the file contains an id, age, and weight of a specific athlete. The program will contain two functions:
  - **`void readAthleteData`** - This function reads in up to 100 lines of data into an array of structs and returns the number of athletes in the datafile.
  - **`int whatAge`** - This function returns the age of the athlete with the given idNumber.
- Declare a struct for each athlete's data
- Create an array of structs to hold all athlete's data
- Write each function described above