Section 6 – Structures

- structures are a method to create a new data type
  - after we learn about structures (a C programming concept) there will be a natural extension to learn about classes (a C++ programming concept)

6.1 What are structures?

We would often like to associate several values with a single item. For example, how can we represent the real and imaginary parts of a complex number?

This can be achieved using a structure:

```
// struct is a keyword that declares a structure
// type definition
// 'ComplexNum' is the identifier of the structure type
struct ComplexNum
{
       double real; // real and imag are 'member names'
       double imag;
}; //use semi-colon to end struct
```

- structure definitions normally occur outside any functions making them globally available.
- members of a structure do not have to be of the same type

For example, store the Student ID#, course percent, and letter grade in a single structure:

```
struct StudentInfo
{
  int  id;
  double percent;
  char grade;
};
```

Structure variables can be declared in the usual fashion for declaring other variables:

```
StudentInfo  student;
```

Structure values are collections of smaller values called member values that can be accessed using a "dot" operator:

```
student.id                    (int type)
student.percent               (double type)
student.letter                (char type)
```

Values can be assigned to member variables by assignment

```
student.id = 123456;
student.percent = 74.6;
student.grade = 'B';
```

or, a method to create and initialize at the same time:

```
StudentInfo student = {123456, 74.6, 'B'};
 // order is important!!
```

6.2 Passing and Returning Structs

- member variables can be used in the same manner other variables (of the same type) can be used.
  - passed into functions (by reference, by value)
  - used as the return type for a function

Example:

```
void set_default_data( StudentInfo &student );
StudentInfo enter_data( );
void display_data( const StudentInfo &student );

int  main()
{
  StudentInfo student;
  set_default_data( student );
  student = enter_data( );
  display_data( student );
  ...
}
```

Pass a struct that will be modified

```
void set_default_data( StudentInfo &stdt )
{
  stdt.id = 100000;
  stdt.percent = 0.0;
  stdt.grade = 'F';
}
```

Return a struct (another method to modify the struct contents)

```
StudentInfo enter_data( )
{
  StudentInfo stdt;
  cin >> stdt.id;
  cin >> stdt.percent;
  cin >> stdt.grade;
  return stdt;
}
```

Pass a struct that will NOT be modified

```
void display_data( const StudentInfo &stdt )
{
  cout << "Student id: " << stdt.id << endl;
  cout << "Student percent: " << stdt.percent
       << endl;
  cout << "Student grade: " << stdt.grade << endl;
}
```

Why 'const' and '&' as a modifier for the formal parameter? ie. why not just use call-by-value?
&: sends the address of the struct variable (as we learned before the midterm)
const: tells the function that the variable can not be modified ie. cannot insert the following into 'display_data':

```
stdt.id = 555555;
```

Why not just send as call-by-value?
- creates a copy of the argument; this can be quite large for struct variables and (later) objects

6.3 Use of Member Names in Different Structures
The same member names can be repeated in different structures:

```
struct Cube
{
  double dimension;
  double volume;
};
struct Sphere
{
  double radius;
  double volume;
};
Cube Box1, Box2;
Sphere Ball1, Ball2;
Box1.volume = 2.3;
Ball1.volume = 3.5;
```

Assign one struct to another (just like variables); contents of the member names are assigned as well

```
Box1 = Box2;
```

6.4 Hierarchial Structures "structure within a structure"

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

struct StudentInfo
{
  int id;
  double percent;
  char letter;
  Date birthday;
};
```

Example:

```
StudentInfo student;
student.birthday.year = 1980;
```

Note how two dot operators are used to access the year from the StudentInfo structure.