Struct

Suppose that you have an problem in which you would like to manipulate some data about a group of individuals. For example, if you are teaching class you might have some scores from students' work that you would like to manipulate.

first name	last name	quiz 1	quiz 2	quiz 3	lab 1	lab 2	lab3	Exam
Alan	Turing	8	7	5	7.8	8.4	6.2	8.1
Ada	Lovelace	10	8	9	9.1	8.5	7.5	8.9
Grace	Hopper	10	10	8	9.5	10.0	8.4	9.4

You might consider making 7similar arrays:

`quiz1[3], quiz2[3], lab1[3], exam[3],` etc.

Each student's information would correspond to one index — 0 for Alan, 1 for Ada, and 2 for Grace. This seems a bit unwieldy, because you have match the corresponding elements from each array to get all the information about single individual. You might consider a 2D array, but then there is a problem that there are several different data types — you can't mix data types in an array.

Struct

It would make the most sense if we could group together all of the different bits of information for a single individual. Essentially, grouping things horizontally rather the vertically in the table from the previous slide. In a sense, this is like a special type of array that allows different types to be grouped together.

Because this is such a common situation in manipulating data — basically we are talking about creating a database — that C allows for the definition of "data containers" that group together different variables that are all related. These are called *structs*.

Structs are simple to set up and use. They can simplify the process of organizing the data in your programs.

The process starts by defining a new variable using the keyword: `struct`.

```
struct studentData {
      int quiz1;
      int quiz2;
      int quiz3;
      double lab1;
      double lab2;
      double lab3;
      double exam;
};
```

This is the *structure tag*. It is optional, but can be useful for declaring variables later.

data members

note the semi-colon here.

This defines a new variable type called `subjectData`. This plays the same role as the `int, double, char`, etc. that we have used since day one. This is an *aggregate* data type, since it consists of a combination of *data members* (`quiz1, quiz2, lab1`, exam etc.)

Note that the above lines simply provide a *template* for the new data structure. No memory has been allocated yet. To do that, we must declare variables, as we have always done in the past.

Declaring variables

There are two ways for declaring variable with structs. First, they can be included in the struct definition.

Note: no tag used here.

```
struct {
      int quiz1;
      int quiz2;
      int quiz3;
      double lab1;
      double lab2;
      double lab3;
      double exam;
} at, al, gh;
```

Three variables are declared. Now memory is allocated, although nothing is stored yet.

Declaring variables

Secondly, variables can be declared separately from the definition.

Now the tag is necessary

```
struct studentData{
      int quiz1;
      int quiz2;
      int quiz3;
      double lab1;
      double lab2;
      double lab3;
      double exam;
};

struct studentData at, al, gh;
```

This is analogous to variable declarations that we have done in the past. Again, memory has been set aside for all of the data, but no values have been stored yet.

Initializing values

The members of the struct can be initialized when the struct variable is declared, similar to the way that an array might be initialized.

```
struct studentData{
      int quiz1;
      int quiz2;
      int quiz3;
      double lab1;
      double lab2;
      double lab3;
      double exam;
};
struct studentData at = { 8, 7, 5, 7.8, 8.4, 6.2, 8.1 };
struct studentData al = { 10, 8, 9, 9.1, 8.5, 7.5, 8.9 };
struct studentData gh = { 10, 10, 8, 9.5, 10.0, 8.4, 9.4 };
```

This declares new struct variables `at, al,` and `gh` and initializes the data members to the values shown. Note that the member types must match up! Alternatively, the data members can be filled in later.

dot operator

To access the individual members of the struct, the *dot operator* is used. Here is bait of code illustrating the use of the "dot". (The struct definition and declarations are some as on previous slides and are not included below.)

```
int main(){

   int qt;
   double x;

   struct studentData at = { 8, 7, 5, 7.8, 8.4, 6.2, 8.1 };
   struct studentData al = { 10, 8, 9, 9.1, 8.5, 7.5, 8.9 };
   struct studentData gh = { 10, 10, 8, 9.5, 10.0, 8.4, 9.4 };

   at.quiz1 = 10;
   x = al.exam;
   qt = gh.quiz1 + gh.quiz2 + gh.quiz3;

   printf( "%d %4.1lf %d\n\n", at.quiz1, x, qt );

   return 0;
}
```

```
10  8.9 28

Program ended with exit code: 0
```

Basically, anything we could do before with a conventional variable can now be done with a data member that is part of a struct.

Initializing values

So an alternative approach to initializing values in the struct would be do it after it is declared using the dot notation.

```
struct studentData{
     int quiz1, quiz2, quiz3;
     double lab1, lab2, lab3, exam;
};

struct studentData at;

at.quiz1 = 8;
at.quiz2 = 7;
at.quiz3 = 5;
at.lab1 = 7.8;
at.lab2 = 8.4;
at.lab3 = 6.2;
at.exam = 8.1;
```

Or the user can enter them using **`scanf()`**. Or use **`rand()`**. Or maybe they are the result of some calculation. Or whatever.

Example (average score of quiz 1)

```c
#include <stdio.h>

struct studentData{
   int quiz1, quiz2, quiz3;
   double lab1, lab2, lab3, exam;
};

int main(){

   double q1Avg;

   struct studentData at = { 8, 7, 5, 7.8, 8.4, 6.2, 8.1 };
   struct studentData al = { 10, 8, 9, 9.1, 8.5, 7.5, 8.9 };
   struct studentData gh = { 10, 10, 8, 9.5, 10.0, 8.4, 9.4 };

   q1Avg = (at.quiz1 + al.quiz1 + gh.quiz1)/3.0;

   printf( "The average score of quiz 1 is %4.2lf.\n\n", q1Avg );

   return 0;
}
```

```
The average score of quiz 1 is 9.33.

Program ended with exit code: 0
```

Example (total score for Alan)

Calculate a total score, if weighting is 33.3% quizzes, 33.3% labs, and 33.3% for the exam.

```
#include <stdio.h>

struct studentData{
   int quiz1, quiz2, quiz3;
   double lab1, lab2, lab3, exam;
};

int main(){

   int quizTotal;
   double labTotal, total;

   struct studentData at = { 8, 7, 5, 7.8, 8.4, 6.2, 8.1 };
   struct studentData al = { 10, 8, 9, 9.1, 8.5, 7.5, 8.9 };
   struct studentData gh = { 10, 10, 8, 9.5, 10.0, 8.4, 9.4 };

   quizTotal = at.quiz1 + at.quiz2 + at.quiz3;
   labTotal = at.lab1 + at.quiz2 + at.quiz3;
   total = 33.3*quizTotal/30.0 + 33.3*labTotal/30.0 + 33.3*at.exam/10.0;

   printf( "The total score for Alan is %4.2lf.\n\n", total );

   return 0;
}
```

```
The total score for Alan is 71.15.

Program ended with exit code: 0
```

Arrays of structs

Declaring an array of structs is identical to the procedure of declaring an array of any of the other standard data types. Declaring an array of structs will reserve a continuous block of memory that will hold all the members of the structs (elements) of the array. Using the same struct definition

```
struct studentData{
      int quiz1, quiz2, quiz3;
      double lab1, lab2, lab3, exam;
};

struct studentData peeps[3];     //an array of 3 structs
```

Giving us 3 identical instances of the struct `subjectData`.

Of course, we would need to initialize all of the members.

Example (average score of quiz 1 – using an array of structs)

```
#include <stdio.h>

struct studentData{
   int quiz1, quiz2, quiz3;
   double lab1, lab2, lab3, exam;
};

int main(){

   int i, numOfPeeps = 3;
   double q1Avg = 0;

   struct studentData peeps[ numOfPeeps ];          //an array of 3 structs
   //Many lines initializing the values of the struct
   //elements are omitted here for brevity.

   for( i = 0; i < 3; i++)
      q1Avg = q1Avg + peeps[i].quiz1;

    printf( "The average score of quiz 1 is %4.2lf.\n\n", q1Avg/numOfPeeps );

   return 0;
}
```

```
The average score of quiz 1 is 9.33.

Program ended with exit code: 0
```

Example (total scores for everyone – using an array of structs)

```c
#include <stdio.h>

struct studentData{
   int quiz1, quiz2, quiz3;
   double lab1, lab2, lab3, exam;

int main(){

   int i, quizTotal, numOfPeeps = 3;
   double labTotal, total;

   struct studentData peeps[ numOfPeeps ];       //an array of 3 structs

   //Many lines initializing the values of the struct
   //elements are omitted here for brevity.

   for( i = 0; i < numOfPeeps; i++){
      quizTotal = peeps[i].quiz1 + peeps[i].quiz2 + peeps[i].quiz3;
      labTotal = peeps[i].lab1 + peeps[i].lab2 + peeps[i].lab3;
      total = 33.3*quizTotal/30.0 + 33.3*labTotal/30.0 + 33.3*peeps[i].exam/10.0;
      printf( "The total score for student %d is %4.2lf.\n\n", i, total );
   }

   return 0;
}
```

```
The total score for student 0 is 74.04.

The total score for student 1 is 87.47.

The total score for student 2 is 93.35.

Program ended with exit code: 0
```

arrays as members of structs

In examining the assembled data types, we see that there are a number of quizzes and a number of labs for each student. If there is only a few of each like in our example, having a separate variable for each individual is probably all right. But if the number of quizzes and labs were much bigger, then it would make more sense to group those into arrays. And that is certainly allowed within a struct — an array can be a "member" of a struct. Let's re-arrange our example struct to use arrays for the quizzes and labs.

```
struct studentData{
   int quiz[3];
   double lab[3], exam;
};
```

```
struct studentData peeps[3];     //an array of 3 structs
```

Now, there is an array of structs. Each struct a two arrays within it. To access an individual element of one of the quiz or lab arrays:

```
x = peeps[1].quiz[2];                //3rd quiz from 2nd student
y = peeps[3].lab[0];                 //1st lab from 3rd student
```

Now we can use our looping techniques to cycles through the various structs in the array (an outer loop) and through the arrays that are members of individual structs. One the next slide the program that calculates total scores, but now using structs with array members. (Again, the many tedious lines needed to initialize all of the data members has been omitted.)

In the example, we can see that the calculations for the quiz and lab totals are done using loops, as used many times in the past. Now the program will scale easily — even if there are 100 quizzes and 100 labs (also known as EE 285 from Hell), the code is virtually the same.

Go through the program and follow the loops and indexes.

```
The total score for student 0 is 74.04.

The total score for student 1 is 87.47.

The total score for student 2 is 93.35.

Program ended with exit code: 0
```

```c
#include <stdio.h>

struct studentData{
   int quiz[3];
   double lab[3], exam;
};

int main(){
   int i, j, quizTotal = 0, numOfPeeps = 3;
   double labTotal = 0, total;

   struct studentData peeps[ numOfPeeps ];

   //Many lines initializing the values of the struct
   //elements are omitted here for brevity.

   for( i = 0; i < numOfPeeps; i++){
      quizTotal = 0;
      labTotal = 0;

      for( j = 0; j < 3; j++ ){
         quizTotal = quizTotal + peeps[i].quiz[j];
         labTotal = labTotal + peeps[i].lab[j];
      }

      total = 33.3*quizTotal/30.0 + 33.3*labTotal/30.0 + 33.3*peeps[i].exam/10.0;
      printf( "The total score for student %d is %4.2lf.\n\n", i, total );
   }

   return 0;
}
```

char and strings

Now that we are using arrays as members of the struct, we might as will include strings. Our program seems a bit impersonal when it refers to the the students as "student 0", "student 1", etc. We can certainly include two character arrays in the struct definition to hold the first and last names.

```
struct studentData{
    char firt[15], last[15];
    int quiz[3];
    double lab[3], exam;
};
```

To simplify the manipulation of the strings, we will use the **<string.h>** library. Then we can **strcpy()** to insert the string literals of the names into the character arrays in the structs. The long list of initializations (which we are never showing in the program listing) would include line like those below to take care of the strings.

```
strcpy( peeps[0].first, "Alan" );
strcpy( peeps[0].last, "Turing" );
strcpy( peeps[1].first, "Ada" );
etc.
```

The total score program, with names included, is on the next slide.

```
The total score for Alan Turing is 74.04.

The total score for Ada Lovelace is 87.47.

The total score for Grace Hopper is 93.35.

Program ended with exit code: 0
```

```c
#include <stdio.h>
#include <string.h>

struct studentData{
   char first[15], last[15];
   int quiz[3];
   double lab[3], exam;
};
int main(){
   int i, j, quizTotal = 0, numOfPeeps = 3;
   double labTotal = 0, total;

   struct studentData peeps[ numOfPeeps ];

   //Many lines initializing the values of the struct
   //elements are omitted here for brevity.

   for( i = 0; i < numOfPeeps; i++){
      quizTotal = 0;
      labTotal = 0;

      for( j = 0; j < 3; j++ ){
         quizTotal = quizTotal + peeps[i].quiz[j];
         labTotal = labTotal + peeps[i].lab[j];
      }

      total = 33.3*quizTotal/30.0 + 33.3*labTotal/30.0 + 33.3*peeps[i].exam/10.0;
      printf( "The total score for %s %s is ", peeps[i].first, peeps[i].last );
      printf( "%4.2lf.\n\n", total );
   }

   return 0;
}
```