Lecture 19

Structures

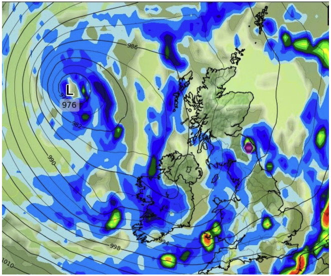
Shibo Li

shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

We have plenty of simple types for storing single items like numbers, characters. But is this really enough for storing more complex things, like patient records, address books, tables, etc.?



Why Structures

It would be easier if we had mechanisms for building up more complex storage items that could be accessed with single variable names

- **user:** `id`, `age`, `hobbies`, `address`, `gender`, ...
- **car:** `color`, `type`, `mgp`, `brand`, `price`, `horse power`, `0-60`, ...
- **smart phone:** `os`, `cpu`, `memory`, `gpu`, `manufacture`, `band`, ...
- ...

Why Structures

- **Compund Storage** - there are some built-in ways to encapsulate multiple pieces of data under one name
  - **Array** - we already know about this one. Indexed collections,and all items are the same type
  - **Sturcture** - keyword struct gives us another way to encapsulate multiple data items into one unit. In this case, items do not have to be the same type
- Structures are good for building records – like database records, or records in a file

What is a Structure

A structure is a collection of data elements, encapsulated into one unit.

- A structure definition is like a blueprint for the structure. It takes up no storage space itself – it just specifies what variables of this structure type will look like
- An actual structure variable is like a box with multiple data fields inside of it. Consider the idea of a student database. One student record contains multiple items of information (name, address, SSN, GPA, etc)

What is a Structure

- Properties of a structure:
  - internal elements may be of various data types
  - order of elements is arbitrary (no indexing, like with arrays)
  - Fixed size, based on the combined sizes of the internal elements

Structure Definitions The basic format of a structure definition is:

```
struct structureName
{
   // data elements in the structure
};
```

- `struct` is a keyword
- The data elements inside are declared as normal variables. `structureName` becomes a new type.
- By themselves, these definitions above are not variables anddo not take up storage

Example

```
/* A structure representing the parts of a fraction (a
    rational number) */
struct Fraction
{
   int num; // the numerator of the fraction
   int denom; // the denominator of the fraction
};
```

Example

```
/* A structure representing a record in a student
    database */
struct Student
{
   char fName[20]; // first name
   char lName[20]; // last name
   int socSecNumber; // social security number double
   gpa; // grade point average
};
```

Structure variables

- To create an actual structure variable, use the structure's name as a type, and declare a variable from it. Format:

```
structureName variableName;
```

- Variations on this format include the usual forms for creating arrays and pointers, and the comma-separated list for multiple variables

Structure variables

- Examples:

```
Fraction f1; // f1 is now a 'Fraction'
Fraction fList[10]; // an array of
    'Fraction'//structures
Fraction * fptr; // a pointer to a
    'Fraction'//structure
Student stu1; // a Student structure variable
Student mathclass[10]; // an array of 10 Students
Student s1, s2, s3; // three Student variables
```

Structure variables

- Examples:

```
Fraction f1; // f1 is now a 'Fraction'
Fraction fList[10]; // an array of
    'Fraction'//structures
Fraction * fptr; // a pointer to a
    'Fraction'//structure
Student stu1; // a Student structure variable
Student mathclass[10]; // an array of 10 Students
Student s1, s2, s3; // three Student variables
```

Legal variations in declaration syntax

- The definition of a structure and the creation of variables canbe combined into a single declaration, as well.
- Just list the variables after the structure definition block (the blueprint), and before the semi-colon:

```
struct structureName
{
   // data elements in the structure
} variable1, variable2, ... , variableN;
```

Example

```
struct Fraction
{
   int num; // the numerator of the fraction
   int denom; // the denominator of the fraction
} f1, fList[10], *fptr; // variable, array, and pointer
    created
```

Legal variations in declaration syntax

- In fact, if you only want structure variables, but don't plan to re-use the structure type (i.e. the blueprint), you don't even need a structure name:

```
struct
// note: no structure NAME given
{int num;
   int denom;
} f1, f2, f3;
// three variables representing fractions
```

- Of course, the advantage of giving a structure definition a name is that it is reusable. It can be used to create structure variables at any point later on in a program, separate from the definition block.

Legal variations in declaration syntax

- You can even declare structures as variables inside of other structure defintions (of different types):

```
struct Date // a structure to represent a date {
   int month;
   int day;
   int year;
};
struct Employee
// a structure to represent an employee of a
    company
{
   char firstName[20];
   char lastName[20];
   Date hireDate;
   Date birthDate;
};
```

Using structures

- Once a structure variable is created, how do we use it? How do we access its internal variables (often known as its members)?
- To access the contents of a structure, we use the dot-operator. Format:

```
structVariableName.dataVariableName
```

- Example, using the fraction structure:

```
Fraction f1, f2;
f1.num = 4; // set f1's numerator to 4
f1.denom = 5; // set f1's denominator to 5
f2.num = 3; // set f2's numerator to 3
f2.denom = 10; // set f2's denominator to 10
cout << f1.num << '/' << f1.denom; // prints 4/5
cout << f2.num << '/' << f2.denom; // prints 3/10
```

- Example, using the fraction structure:

```
Fraction f1, f2;
f1.num = 4; // set f1's numerator to 4
f1.denom = 5; // set f1's denominator to 5
f2.num = 3; // set f2's numerator to 3
f2.denom = 10; // set f2's denominator to 10
cout << f1.num << '/' << f1.denom; // prints
    4/5
cout << f2.num << '/' << f2.denom; // prints
    3/10
```

Example, using the student structure:

```
Student sList[10]; // array of 10 students
// set first student's data: (John Smith, SSN:123456789,
    GPA: 3.75)

strcpy(sList[0].fName, "John");
strcpy(sList[0].lName, "Smith");
sList[0].socSecNumber = 123456789;
sList[0].gpa = 3.75;
```

Example, using the student structure:

```
// assume there's more code here that initializes other
    students
// This loop prints all 10 students -- their names and
    their GPA
cout << fixed << setprecision(2);
for (int i = 0; i < 10; i++)
{
   cout << sList[i].fName << ' ' << sList[i].lName << ' '
       << sList[i].gpa << '\n';
}
```

A shortcut for initializing structs

- While we can certainly initialize each variable in a structureseparately, we can use an initializer list on the declaration line, too
- This is similar to what we saw with arrays
- This is only usable on the declaration line (like with arrays)
- The initializer set should contain the struct contents in thesame order that they appear in the struct definition

Examples

```
Fraction f1 = 3, 5; //initialize num=3, denom=5 // This
    would be the same as doing the following:
f1.num = 3;
f1.denom = 5;
```

```
Student s1 = {"John", "Smith", 123456789, 3.75};
Student s2 = {"Alice", "Jones", 123123123, 2.66};
```

- If we have a pointer to a structure, things are a little trickier:

```
Fraction f1; // a fraction structure
Fraction *fPtr; // pointer to a fraction
fPtr = &f1; // fPtr now points to f1
f1.num = 3; // this is legal, of course
fPtr.num = 10; // how about this?
```

Use Pointers to Structure

- If we have a pointer to a structure, things are a little trickier:

```
Fraction f1; // a fraction structure
Fraction *fPtr; // pointer to a fraction
fPtr = &f1; // fPtr now points to f1
f1.num = 3; // this is legal, of course
fPtr.num = 10; // how about this?
```

- **NO! ILLEGAL, cannot put a pointer on the left side of the dot-operator**

Use Pointers to Structure

- Remember that to get to the target of a pointer, we dereference it. The target of `fPtr` is `*fPtr`. So how about this?

```
*fPtr.num = 10; // how about this?
```

Use Pointers to Structure

```
*fPtr.num = 10; // how about this?
```

- **Closer but not quite.**
- The problem with this is that the dot-operator has higher precedence, so this would be interpreted as:

```
*(fPtr.num) = 10; // cannot put a pointer on the
    left of the dot
```

- But if we use parentheses to force the dereference to happen first, then it works:

```
(*fPtr).num = 10; // YES!
```

- Alternative operator for pointers: While the above exampleworks, it's a little cumbersome to have to use the parentheses and the dereference operator all the time.
- So there is a special operator for use with pointers to structures. It is the arrow operator:

```
pointerToStruct -> dataVariable
```

Example

```
Fraction * fPtr; // pointer to a fraction
// assume this has been pointed at a valid target
fPtr->num = 10; // set fraction's numerator to 10
fPtr->denom = 11; // denominator set to 11
cout << fPtr->num << '/' << fPtr->denom;
//prints: 10/11
```

Accessing members of nested structures

- Earlier, we saw an example of a structure variable used within another structure definition

```
struct Date // Date is now a type name
{
   int month;
   int day;
   int year;
}; // so that "Date" is the type name
struct Employee
{
   char firstName[20];
   char lastName[20];
   Date hireDate;
   Date birthDate;
};
```

Accessing members of nested structures

Here's an example of initializing all the data elements for one employee variable:

```
Employee emp; // emp is an employee variable

// Set the name to "Alice Jones"
strcpy(emp.firstName, "Alice");
strcpy(emp.lastName, "Jones");

// set the hire date to March 14, 2001
emp.hireDate.month = 3;
emp.hireDate.day = 14;
emp.hireDate.year = 2001;

// sets the birth date to Sept 15, 1972
emp.birthDate.month = 9;
emp.birthDate.day = 15;
emp.birthDate.year = 1972;
```

Here's an example of an employee initialization using our shortcut initializer form:

```
Employee emp2 = { "John", "Smith", {6, 10, 2003},{2,
    19, 1981} };

// John Smith, whose birthday is Feb 19, 1981, was
    hired on June 10, 2003
```

- With regular primitive types we have a wide variety ofoperations available, including assignment, comparisons, arithmetic, etc.
- Most of these operations would NOT make sense on structures. Arithmetic and comparisons, for example:

```
Student s1, s2;
s1 = s1 + s2; // ILLEGAL!
// How would we add two students, anyway?
if (s1 < s2) // ILLEGAL. What would this mean?
```

Structures and the assignment operator

- Using the assignment operator on structures IS legal, as long as they are the same type. Example (using previous struct definitions):

```
Student s1, s2;
Fraction f1, f2;
s1 = s2; // LEGAL. Copies contents of s2 into s1
f1 = f2; // LEGAL. Copies f2 into f1
```

Structures and the assignment operator

- Note that in the above example, the two assignment statements are equivalent to doing the following:

```
// these 4 lines are equivalent to s1 = s2;
strcpy(s1.fName, s2.fName);
strcpy(s1.lName, s2.lName);
s1.socSecNumber = s2.socSecNumber;
s1.gpa = s2.gpa;

//these 2 lines are equivalent to f1 = f2;
f1.num = f2.num;
f1.denom = f2.denom;
```

- Clearly, direct assignment between entire structures is easier, if a full copy of the whole thing is the desired result!

Passing structures into and out of functions

- Just like a variable of a basic type, a structure can be passed into functions, and a structure can be returned from a function.
- To use structures in functions, use `structname` as theparameter type, or as a return type, on a function declaration

```
// function that passes a structure variable as a
    parameter
void PrintStudent(Student s);

// function that passes in structure variables and
    returns a struct
Fraction Add(Fraction f1, Fraction f2);
```

Pass by value, reference, address

- Just like with regular varaibles, structures can be passed byvalue or by reference, or a pointer to a structure can be passed (i.e., pass by address)
- If just a plain structure variable is passed, as in the aboveexamples, it's pass by value. A copy of the structure is made
- To pass by reference, use the & on the structure type, just as with regular data types

Pass by value, reference, address

- To pass by address, use pointers to structures as theparameters and/or return
- As with pointers to the built-in types, you can use const toensure a function cannot change the target of a pointer
- It's often a GOOD idea to pass structures to and from functions by address or by reference
  - structures are compound data, usually larger than plain atomic variables
  - Pass-by-value means copying a structure. NOT copying is desirable for efficiency, especially if the structure is very large

Examples

```
// function that passes a pointer to student
//structure as a parameter
void GetStudentData(Student* s);

// function that passes in structures by const
// reference, and returns a struct by value
Fraction Add(const Fraction& f1, const Fraction& f2);
```

Examples

```
//function that uses const on a structure pointer
// parameter. This function could take in an array
// of Students, or the address of one student.
void PrintStudents(const Student* s);

// or, this prototype is equivalent to the one above
void PrintStudents(const Student s[]);
```