Basis and Practice in Programming

Chapter 9: Structures

Prof. Tamer ABUHMED
College of Software

성균관대학교
SUNG KYUN KWAN UNIVERSITY(SKKU)

Lecture Objectives

- Structures [chapter 9]
  - Defining and using Structures
  - Functions and Structures
  - Initializing Structures. Compound Literals
  - Arrays of Structures
  - Structures Containing Structures and/or Arrays
- More on Data Types [Chapter 14]
  - Enumerated Data Types
  - The typedef Statement
  - Data Type Conversions

The concept of structures

- Structure: a tool for grouping heterogenous elements together.
- Array: a tool for grouping homogenous elements together

- Example: storing calendar dates (day, month, year)
- Version1: using independent variables:
- int month = 9, day = 25, year = 2004;

- Using this method, you must keep track of three separate variables for each date that you use in the program—variables that are logically related. It would be much better if you could somehow group these sets of three variables together. This is what the structure in C allows you to do !

Example: structures

```c
struct date
{
    int month;
    int day;
    int year;
};
```

Defines type `struct date`, with 3 ***fields*** of type `int`
The names of the fields are local in the context of the structure.
A struct declaration defines a type: if not followed by a list of variables it reserves no storage; it merely describes a template or shape of a structure.

```c
struct date today, purchaseDate;
```

Use 3 variables of type `struct date`

```c
today.year = 2004;
today.month = 10;
today.day = 5;
```

Accesses fields of a variable of type `struct date`
A member of a particular structure is referred to in an expression by a construction of the form `structurename.member`

Example: determine tomorrow's date (Version 1)

```c
// Program to determine tomorrow's date
#include <stdio.h>
int main (void)
{
     struct date
     {
          int month;
          int day;
          int year;
     };
     struct date today, tomorrow;
     const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
     31, 31, 30, 31, 30, 31 };

     printf ("Enter today's date (mm dd yyyy): ");
     scanf ("%i%i%i", &today.month, &today.day, &today.year);
```

Example continued

```
if ( today.day != daysPerMonth[today.month - 1] ) {
     tomorrow.day = today.day + 1;
     tomorrow.month = today.month;
     tomorrow.year = today.year;
}
else if ( today.month == 12 ) { // end of year
     tomorrow.day = 1;
     tomorrow.month = 1;
     tomorrow.year = today.year + 1;
}
else { // end of month
     tomorrow.day = 1;
     tomorrow.month = today.month + 1;
     tomorrow.year = today.year;
}
printf ("Tomorrow's date is %i/%i/%i.\n", tomorrow.month,
                                     tomorrow.day, tomorrow.year );
return 0;
}
```

Output1

```
Enter today's date (mm dd yyyy): 03 05 1999
Tomorrow's date is 3/6/1999.
```

Output2

```
Enter today's date (mm dd yyyy): 02 28 2004
Tomorrow's date is 3/1/2004.
```

Operations on structures

- Legal operations on a structure are :
  - copying it or assigning to it as a unit
    - this includes passing arguments to functions and returning values from functions as well.
  - taking its address with &
  - accessing its members.
  - structures may **not** be compared as units !
  - a structure may be initialized by a list of constant member values

Example: determine tomorrow's date (Version 2)

```
// Program to determine tomorrow's date
#include <stdio.h>
#include <stdbool.h>

struct date
{
     int month;
     int day;
     int year;
};
int numberOfDays (struct date d);

int main (void)
{
struct date today, tomorrow;
printf ("Enter today's date (mm dd yyyy): ");
scanf ("%i%i%i", &today.month, &today.day, &today.year);
```

Defines type struct date as a global type

Declares a function that takes a struct date as a parameter

Example continued

```
if ( today.day != numberOfDays (today) ) {
     tomorrow.day = today.day + 1;
     tomorrow.month = today.month;
     tomorrow.year = today.year;
}
else if ( today.month == 12 ) { // end of year
     tomorrow.day = 1;
     tomorrow.month = 1;
     tomorrow.year = today.year + 1;
}
else { // end of month
     tomorrow.day = 1;
     tomorrow.month = today.month + 1;
     tomorrow.year = today.year;
}
printf ("Tomorrow's date is %i/%i/%i.\n",tomorrow.month,
tomorrow.day, tomorrow.year);
return 0;
}
```

Example continued

```c
bool isLeapYear (struct date d);

// Function to find the number of days in a month
int numberOfDays (struct date d)  {
     int days;
     const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
     if ( isLeapYear (d) == true && d.month == 2 )
          days = 29;
     else
          days = daysPerMonth[d.month - 1];
     return days;
}

// Function to determine if it's a leap year
bool isLeapYear (struct date d) {
     bool leapYearFlag;
     if ( (d.year % 4 == 0 && d.year % 100 != 0) || d.year % 400 == 0 )
          leapYearFlag = true; // It's a leap year
     else
          leapYearFlag = false; // Not a leap year
     return leapYearFlag;
}
```

Output1

```
Enter today's date (mm dd yyyy): 02 28 2020
Tomorrow's date is 2/29/2020.
```

Output2

```
Enter today's date (mm dd yyyy): 10 30 2020
Tomorrow's date is 10/31/2020.
```

Example: determine tomorrow’s date (Version 3)

```c
// Program to determine tomorrow's date
#include <stdio.h>
#include <stdbool.h>
struct date
{
     int month;
     int day;
     int year;
};

struct date dateUpdate (struct date today);

int main (void){
     struct date thisDay, nextDay;
     printf ("Enter today's date (mm dd yyyy): ");
     scanf ("%i%i%i", &thisDay.month, &thisDay.day, &thisDay.year);
     nextDay = dateUpdate (thisDay);
     printf ("Tomorrow's date is %i/%i/%i.\n",nextDay.month,
                                                  nextDay.day, nextDay.year );
     return 0;
}
```

Declares a function that takes a struct date as a parameter and returns a struct date

Example continued

```c
int numberOfDays (struct date d);

// Function to calculate tomorrow's date
struct date dateUpdate (struct date today) {
     struct date tomorrow;
     if ( today.day != numberOfDays (today) ) {
          tomorrow.day = today.day + 1;
          tomorrow.month = today.month;
          tomorrow.year = today.year;
     }
     else if ( today.month == 12 ) { // end of year
          tomorrow.day = 1;
          tomorrow.month = 1;
          tomorrow.year = today.year + 1;
     }
     else { // end of month
          tomorrow.day = 1;
          tomorrow.month = today.month + 1;
          tomorrow.year = today.year;
     }
     return tomorrow;
}
```

Output1

```
Enter today's date (mm dd yyyy): 02 28 2020
Tomorrow's date is 2/29/2020.
```

Output2

```
Enter today's date (mm dd yyyy): 10 30 2020
Tomorrow's date is 10/31/2020.
```

Example: Update time by one second

```
// Program to update the time by one second
#include <stdio.h>
struct time
{
     int hour;
     int minutes;
     int seconds;
};

struct time timeUpdate (struct time now);

int main (void) {
     struct time currentTime, nextTime;
     printf ("Enter the time (hh:mm:ss): ");
     scanf ("%i:%i:%i", &currentTime.hour, &currentTime.minutes,
                                        &currentTime.seconds);
     nextTime = timeUpdate (currentTime);
     printf ("Updated time is %.2i:%.2i:%.2i\n", nextTime.hour,
                              nextTime.minutes, nextTime.seconds );
     return 0;
}
```

Example continued

```c
// Function to update the time by one second
struct time timeUpdate (struct time now)  {
     ++now.seconds;
     if ( now.seconds == 60 ) { // next minute
          now.seconds = 0;
          ++now.minutes;
          if ( now.minutes == 60 ) { // next hour
                    now.minutes = 0;
                    ++now.hour;
                    if ( now.hour == 24 ) // midnight
                              now.hour = 0;
                    }
          }
     return now;
}
```

Parameters of a struct type are passed by copy of value

Initializing structures. Compound literals

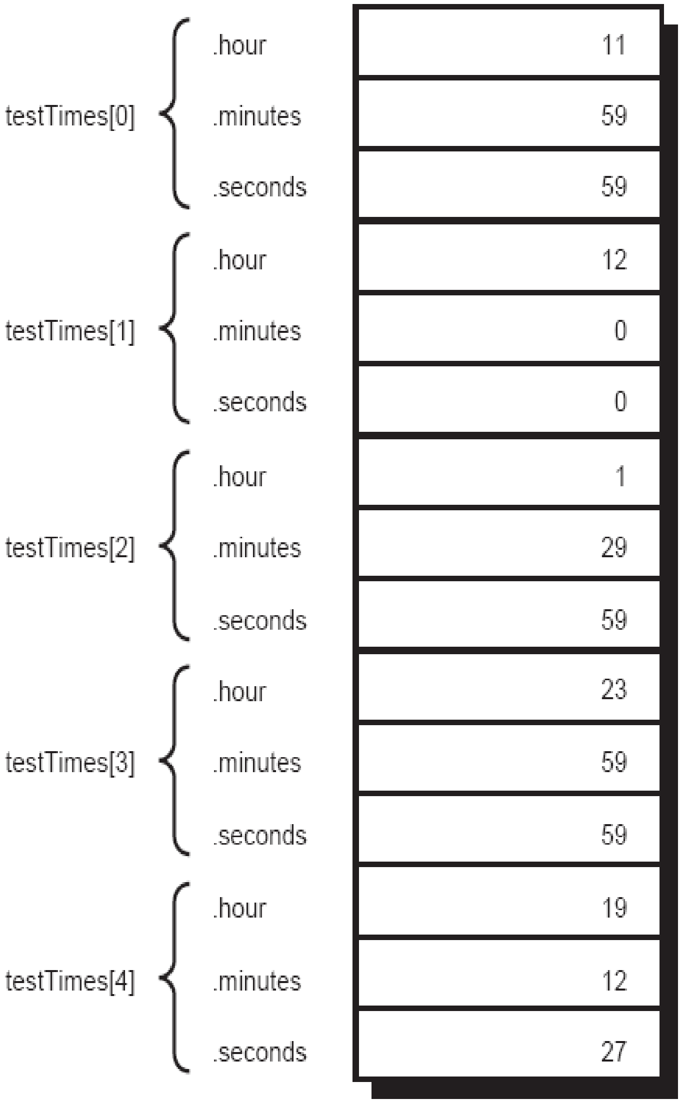
```
struct date today = { 7, 2, 2005 };
struct time this_time = { 3, 29, 55 };
today = (struct date) { 9, 25, 2004 };
today = (struct date) { .month = 9, .day = 25, .year = 2004 };
```

Arrays of structures

```
struct time {  . . .  };

struct time timeUpdate (struct time now);

int main (void) {
     struct time testTimes[5] =
     { { 11, 59, 59 }, { 12, 0, 0 }, { 1, 29, 59 },
       { 23, 59, 59 }, { 19, 12, 27 }};
     int i;
     for ( i = 0; i < 5; ++i ) {
          printf ("Time is %.2i:%.2i:%.2i", testTimes[i].hour,
                  testTimes[i].minutes,  testTimes[i].seconds);
          testTimes[i] = timeUpdate (testTimes[i]);
          printf (" ...one second later it's %.2i:%.2i:%.2i\n",
                   testTimes[i].hour, testTimes[i].minutes,
                   testTimes[i].seconds);
          }
     return 0;
}
```


testTimes[0]
.hour
11
.minutes
59
.seconds
59
testTimes[1]
.hour
12
.minutes
0
.seconds
0
testTimes[2]
.hour
1
.minutes
29
.seconds
59
testTimes[3]
.hour
23
.minutes
59
.seconds
59
testTimes[4]
.hour
19
.minutes
12
.seconds
27

Structures containing structures

```c
struct dateAndTime
{
     struct date sdate;
     struct time stime;
};

. . .

struct dateAndTime event;

event.sdate = dateUpdate (event.sdate);

event.sdate.month = 10;
```

Structures containing arrays

```
struct month
{
     int numberOfDays;
     char name[3];
};

struct month aMonth;

. . .

aMonth.numberOfDays = 31;
aMonth.name[0] = 'J';
aMonth.name[1] = 'a';
aMonth.name[2] = 'n';
```

Enumerated data type

- You can use the enumerated type to declare symbolic names to represent integer constants.
- By using the `enum` keyword, you can create a new "type" and specify the values it may have.
- Actually, enum constants are type int; therefore, they can be used wherever you would use an int.
- The purpose of enumerated types is to enhance the readability of a program.

```
enum primaryColor { red, yellow, blue };
enum primaryColor myColor, gregsColor;
myColor = red;
if ( gregsColor == yellow ) …
```

Enumerated data type

- The C compiler actually treats enumeration identifiers as integer constants. Beginning with the first name in the list, the compiler assigns sequential integer values to these names, starting with 0.

```
enum month thisMonth;
...
thisMonth = february;
```

- The value 1 is assigned to `thisMonth` because it is the second identifier listed inside the enumeration list.
- If you want to have a specific integer value associated with an enumeration identifier, the integer can be assigned to the identifier when the data type is defined. Enumeration identifiers that subsequently appear in the list are assigned sequential integer values beginning with the specified integer value plus 1. For example, in the definition

```
enum direction { up, down, left = 10, right };
```

- An enumerated data type direction is defined with the values up, down, left, and right. The compiler assigns the value 0 to up because it appears first in the list; 1 to down because it appears next; 10 to left because it is explicitly assigned this value; and 11 to right because it appears immediately after left in the list.

Example: enum

```
// Program to print the number of days in a month
#include <stdio.h>
int main (void) {
     enum month { january = 1, february, march, april, may, june,july, august, september,
     october, november, december };
     enum month aMonth;
     int days;
     printf (“Enter month number: “);
    scanf (“%i”, &aMonth);
    switch (aMonth ) {
       case january: case march: case may: case july:
       case august: case october: case december:
          days = 31;  break;
       case april: case june: case september: case november:
          days = 30;  break;
       case february:
          days = 28;  break;
       default:
          printf (“bad month number\n”);
          days = 0;   break;
  }
   if ( days != 0 )
         printf (“Number of days is %i\n”, days);
   if ( aMonth == february )
         printf (“...or 29 if it’s a leap year\n”);
return 0;
}
```

The `typedef` statement

- C provides a capability that enables you to ***assign an alternate name to a data type***. This is done with a statement known as `typedef`.

```
typedef type_description type_name;
```

- The statement

```
typedef int Counter;
```

- defines the name `Counter` to be equivalent to the C data type `int`. Variables can subsequently be declared to be of type `Counter`, as in the following statement:

```
Counter j, n;
```

- The C compiler actually treats the declaration of the variables j and n, shown in the preceding code, as normal integer variables.
- The main advantage of the use of the `typedef` in this case is in the added **readability** that it lends to the definition of the variables.
- the `typedef` statement ***does not actually define a new type—only a new type name.***

The `typedef` statement

- In forming a typedef definition, proceed as though a normal variable declaration were being made. Then, place the new type name where the variable name would normally appear. Finally, in front of everything, place the keyword typedef:

```
typedef char Linebuf [81];
```

- defines a type called Linebuf, which is an array of 81 characters. Subsequently declaring variables to be of type Linebuf, as in

```
Linebuf text, inputLine;

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


Date birthdays[100];
```

Data type conversions

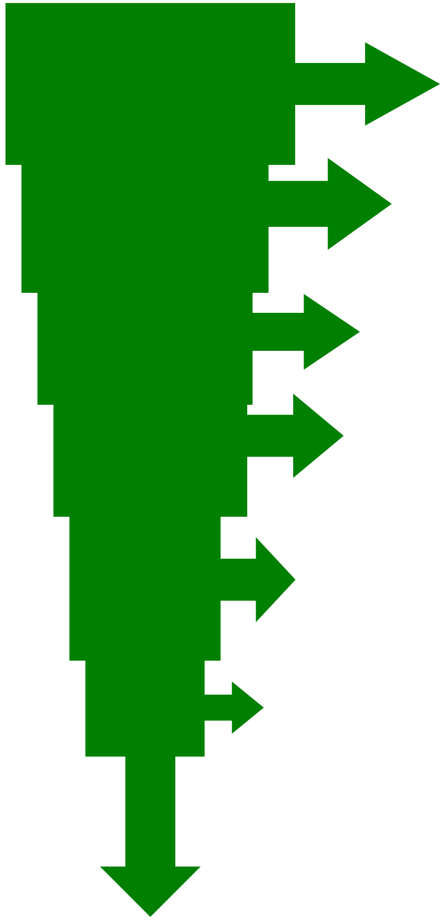
- *Expressions: Operands and operators*
- *Operands may be of different types; issue: how is the expression evaluated and what is the type of the result ?*

- sometimes conversions are implicitly made by the system when expressions are evaluated !
- The C compiler adheres to strict rules when it comes to evaluating expressions that consist of different data types.
- **Essence of *automatic conversion rules*: convert "smaller" type to "bigger" type**

- Example: the case examined in lecture 2 was with the data types `float` and `int`: an operation that involved a float and an int was carried out as a floating-point operation, ***the integer data item being automatically converted to floating point***.

- the ***type cast operator*** **can be used to explicitly dictate a conversion.**

Rules for automatic conversions



1. If either operand is of type `long double`, the other is converted to long double, and that is the type of the result.
2. If either operand is of type `double`, the other is converted to double, and that is the type of the result.
3. If either operand is of type `float`, the other is converted to float, and that is the type of the result.
4. If either operand is of type `_Bool`, `char`, `short int`, or of an `enumerated` data type, it is converted to int.
5. If either operand is of type `long long int`, the other is converted to long long int, and that is the type of the result.
6. If either operand is of type `long int`, the other is converted to long int, and that is the type of the result.
7. If this step is reached, both operands are of type int, and that is the type of the result.

Example: f is defined to be a float, i an int, l a long int, and s a short int variable: evaluate expression `f * i + l / s`

Sign extensions

- Conversion of a signed integer to a longer integer results in extension of the sign (0 or 1) to the left;
  - This ensures that a short int having a value of –5 will also have the value –5 when converted to a long int.
- Conversion of an unsigned integer to a longer integer results in zero fill to the left.

Signed & Unsigned types

- If in an expression appear both signed and unsigned operands, **signed operands are automatically converted to unsigned**
- Converting signed to unsigned:
  - No change in bit representation
  - Nonnegative values unchanged
  - Negative values change into positive values !
  - Example: `int x=-5; unsigned int ux=(unsigned int) x;` `printf("%ud \n",ux);` // 4294966*62
- Conversion surprises:

```
int a=-5;
 unsigned int b=1;

 if (a>b)
   printf("a is bigger than b");
 else if (a<b)
   printf("b is bigger than a");
 else
   printf("a equals b");
```

Explicit conversions/casts

- Rules:
  - Conversion of any value to a _Bool results in 0 if the value is zero and 1 otherwise.
  - Conversion of a longer integer to a shorter one results in truncation of the integer on the left.
  - Conversion of a floating-point value to an integer results in truncation of the decimal portion of the value. If the integer is not large enough to contain the converted floating-point value, the result is not defined, as is the result of converting a negative floating-point value to an unsigned integer.
  - Conversion of a longer floating-point value to a shorter one might or might not result in rounding before the truncation occurs.

Lecture Summary

- Structures [chapter 9]
  - Defining and using Structures
  - Functions and Structures
  - Initializing Structures. Compound Literals
  - Arrays of Structures
  - Structures Containing Structures and/or Arrays
- More on Data Types [Chapter 14]
  - Enumerated Data Types
  - The typedef Statement
  - Data Type Conversions