Enumeration Types and Structures

CSE 130: Introduction to Programming in C

Stony Brook University

Enumeration Types

Enumeration Types

- Used to:
  - name a finite set
  - declare elements of that set (*enumerators*)
- Used as programmer-specified constants
- Ex. `enum color {red, blue, green, yellow};`
  - `color` is the *tag name*

Enumerators

- *Enumerators* specify the values that variables of the enumerated type can take on
  - Ex. `enum boolean {false, true};`
- These are constants of type `int`
  - By default, they are given the values 0, 1, ...
  - They can also be assigned specific values

Enumeration Type Variables

- Ex. `enum color c1, c2;`
    - `c1` and `c2` are of type `enum color`
    - Note: the type is `enum color`, *NOT* `color`
    - `c1` and `c2` can *only* take on the values red, blue, green, and yellow:

```
c1 = green;
```

Initializing Enumerators

- `enum suit {clubs = 1, diamonds, hearts, spades};`
    - `diamonds`, `hearts`, and `spades` have the values 2, 3, and 4 respectively
- Uninitialized enumerators are assigned consecutive values, starting after the last initialized enumerator
- The values may be duplicated, but the identifiers must be unique

More Declaration Examples

- `enum suit {clubs, diamonds, hearts, spades} a;`
  - `a` is of type `enum suit`
- If we omit the tag name, then every variable of that type must be declared as part of the enumeration type:
  - `enum {fir, pine} tree;`
    - No other variables of type `enum {fir, pine}` can be declared

```
enum move {rock, paper, scissors};
enum outcome {win, lose, tie};
...
enum outcome result;
if (player == computer)
    result = tie;
else
{
    switch(player)
    {
        case paper:
            result = (computer == rock) ? win : lose;
            break;
        case scissors:
            result = (computer == paper) ? win : lose;
            break;
        etc.
    }
}
```

Structures

The Structure Type

- A ***structure*** makes it possible to aggregate components into a single, named variable
  - Ex. a bank account contains an account #, a balance, an interest rate, etc.
- Structure components have individual names, and can be accessed individually
- A structure is a *derived type*
- It's sort of like a primitive/limited class from an object-oriented language

Declaring a Structure

- Structure declarations begin with the keyword `struct`, followed by a tag name and a brace-enclosed list of components
- The tag name can be used to declare variables of the structure's type
  - The variable type is `struct tag-name`

Structure Example

```
struct account /* tag name is account */
{
     long number;
     float balance;
     float interestRate;
};

struct account myAcct;
```

Structure Members

- Members of a structure can be accessed using the structure member (".") operator:

```
struct account a;
a.balance = 1234.56;
a.number = 8463745;
```

- Member names must be unique within the same structure
- Two different structure types may have identical member names, though

Structure Declarations

- We can combine a structure definition with variable declarations
- ```
  struct card
  {
      int value;
      char suit;
  } c, deck[52];
  ```
  ```

Structure Example 2a

```
struct fruit
{
    char name[15];
    int calories;
};

struct vegetable
{
    char name[15];
    int calories;
};
```

Structure Example 2b

```
struct fruit a;
struct vegetable b;
a.calories = 35;
b.calories = 45;
```

Another Example

```
struct student
{
    char *lastName;
    int studentID;
    char grade;
};
```

```
int fail(struct student class[])
{
    int i, count = 0;
    for (i = 0; i < CLASS_SIZE; i++)
        if (class[i].grade == 'F')
            count++;
    return count;
}
```

Structure Initialization

- A structure variable can be followed by a list of constants contained within braces
    - the remaining members are assigned the value 0
    - Ex. `struct card c = {12, 's'};`
    - Ex. `struct fruit frt = {"plum", 150};`
- We can also name members, as with arrays:

```
struct card c = {.value = 5, .suit = 'd'};
```

Structure Assignment

- If `a` and `b` are variables of the same structure type, we can write

```
a = b;
```

- Each member of `a` is assigned the value of the corresponding value of `b`

Passing Structures As Function Arguments

```
void assignValues(struct card c, int p,
                                  char s)
{
   c.value = p;
   c.suit = s;
}
```

Passing Structures

- When a structure is passed as an argument, it is copied (because of call-by-value)
- It is more efficient to pass the address of the structure instead
- In this case, use the ***member access operator* –>** (a dash followed by an arrow bracket) to manipulate the structure's members:

```
p -> data = 25;
```

Example: Member Access

Declaration and Assignment		

struct student tmp, *p = &tmp;
tmp.grade = ′A′;
tmp.last_name = ″Casanova″;
tmp.student_id = 910017;		

Expression	Equivalent Expression	Conceptual Value
tmp.grade	p->grade	A
tmp.last_name	p->last_name	Casanova
(*p).student_id	p->student_id	910017
*p->last_name+1	(*(p->last_name))+1	D
*(p->last_name + 2)	(p->last_name)[2]	s

Using Structures with Functions

- Structures can be passed as arguments to a function and can be returned from them.
- When a structure is passed as an argument to a function, it is passed by value, meaning that a local copy is made for use in the body.
  - If a member of the structure is an array, then the array gets copied as well.
  - If the structure has many members, or members that are large arrays, then passing the structure as an argument can be relatively inefficient.
- An alternate scheme is to write functions that take an address of the structure as an argument instead.

Example: Business Application

```
struct dept {
   char dept_name[25];
   int dep_no;
} ;
```

the compiler has to know the size of each member

```
typedef struct {

   char name[25];

   int employee_id;

   struct dept department;

   struct home_address *a_ptr;

   double salary;

} employee_data;
```

Structure type member

Pointer to a Structure

the compiler already knows the size of a pointer, this structure need not be defined first.

Example: Business Application

- Function to update employee information

```
employee_data update(employee_data e)
{
    printf("Input the department number: ");
    scanf("%d", &n);
    e.department.dept_no = n;
    return e;
}
```

  - we are accessing a member of a structure within a structure

`e.department.dept_no` is equivalent to
`(e.department).dept_no`

- To use the function `update()`, we could write in `main()` or in some other function

```
employee_data e;

e = update(e);
```

Copy Problem

```
employee_data update(employee_data e)
{
    printf("Input the department number: ");
    scanf("%d", &n);
    e.department.dept_no = n;
    return e;
}
```

```
employee_data e;

e = update(e);
```

- `e` is being passed by value, causing a local copy of `e` to be used in the body of the function; when a structure is returned from `update()`, it is assigned to `e`, causing a member-by-member copy to be performed. Because the structure is large, the compiler must do a lot of copy work.

Alternate: Update Function

```
void update(employee_data *p)
{
    printf("Input the department number: ");
    scanf("%d", &n);
    p->department.dept_no = n;
}
```

`p->department.dept_no` is equivalent to `(p->department).dept_no`

This version of update() can be used in main() as follows:

```
employee_data e;

update(&e);
```

- Here, the address of e is being passed, so no local copy of the structure is needed within the update() function. For most applications this is the more efficient of the two methods.