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 = (machine == rock) ? win : lose;
            break;
        case scissors:
            result = (machine == 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;
```