Lecture 15: Structs, Unions, and Enumerations

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology
Winter 2018

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]

Agenda

- Structure variables
- Structure types
- Nested structures
- Unions
- Enumerations

Readings

- Textbook Chapter 16

Structures

Structures

- A *structure* (keyword struct) is a data structure that may have more than one *member.*

- A member is basically an “element” of this structure. Different from arrays, members/elements of a structure can have different types.

- Also different from arrays, “elements” of a structure are accessed by their name, instead of index.

- Useful when you need to store a collection of related data items.

- Structures are the closest thing to a class in object-oriented languages.

Structure Variables

Structure Variables

```
struct{
  int age;
  char name[20];
  char gender;
} person1, person2;
```

Structure Variables

```
struct{
  int age;
  char name[20];
  char gender;
} person1, person2;
```

- This declares two variables `person1` and `person2`, each of type `struct { int age; …}`.

Structure Variables

```
struct{
  int age;
  char name[20];
  char gender;
} person1, person2;
```

- This declares two variables `person1` and `person2`, each of type `struct { int age; …}`.
- The specified struct type has three members: `age`, `name`, and `gender`

Structure Variables

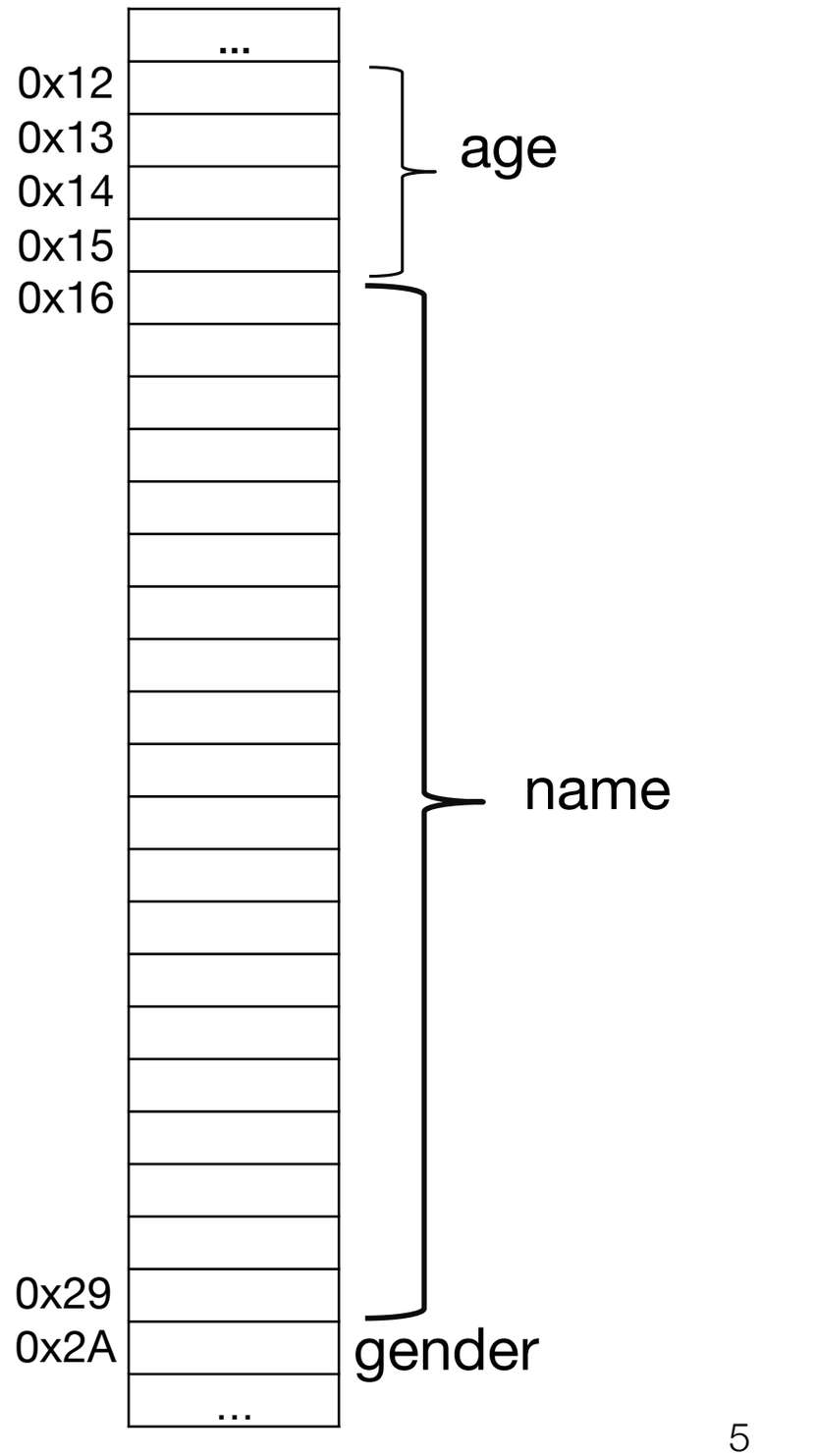
```
struct{
  int age;
  char name[20];
  char gender;
} person1, person2;
```

- This declares two variables `person1` and `person2`, each of type `struct { int age; …}`.
- The specified struct type has three members: `age`, `name`, and `gender`
- Members of a struct are stored in memory in the order they are declared

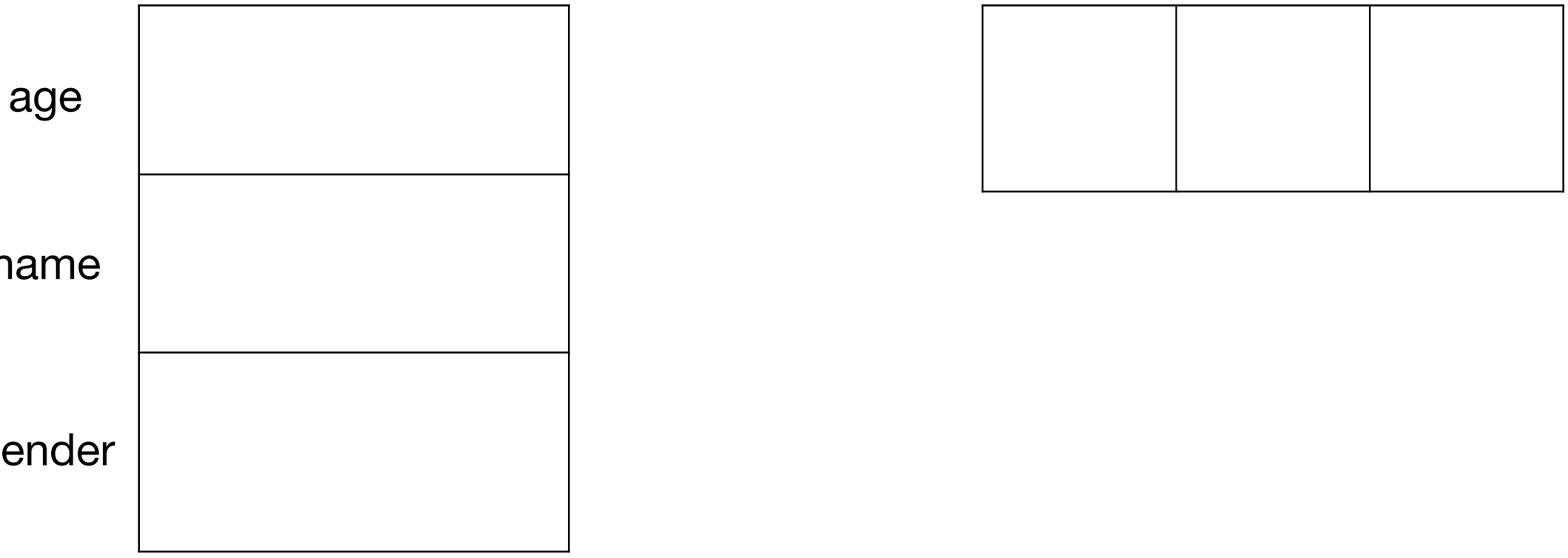
Structure Variables

```
struct{
  int age;
  char name[20];
  char gender;
} person1, person2;
```

- This declares two variables `person1` and `person2`, each of type `struct { int age; …}`.
- The specified struct type has three members: `age`, `name`, and `gender`
- Members of a struct are stored in memory in the order they are declared



More Abstract Visualization of Structs



More on Structs

- Each structure represents a new scope: any identifiers declared in that scope will not conflict with other names in a program

More on Structs

- Each structure represents a new scope: any identifiers declared in that scope will not conflict with other names in a program

```
struct{
  int age;
  char name[20];
  char gender;
} person1, person2;
```

More on Structs

- Each structure represents a new scope: any identifiers declared in that scope will not conflict with other names in a program

```
struct{
  int age;
  char name[20];
  char gender;
} person1, person2;
```

```
struct{
  int number;
  char name[20];
  int available;
} part1, part2;
```

More on Structs

- Each structure represents a new scope: any identifiers declared in that scope will not conflict with other names in a program

different variables
&
both structs can appear in the same scope

```
struct{
  int age;
  char name[20];
  char gender;
} person1, person2;
```

```
struct{
  int number;
  char name[20];
  int available;
} part1, part2;
```

Initializing Struct Variables

Initializing Struct Variables

```
struct{
  int age;
  char name[20];
  char gender;
} person1 = {23, “Bob”, ‘M’},
  person2 = {23, “Alice”, ‘F’};
```

Initializing Struct Variables

```
struct{
  int age;
  char name[20];
  char gender;
} person1 = {23, “Bob”, ‘M’},
  person2 = {23, “Alice”, ‘F’};
```

Note how the order of values matches the order of members in the struct

Initializing Struct Variables

```
struct{
  int age;
  char name[20];
  char gender;
} person1 = {23, “Bob”, ‘M’},
  person2 = {23, “Alice”, ‘F’};
```

Note how the order of values matches the order of members in the struct

- Expressions used in structure initializers must be constant
- An initializer can have fewer values than the number of members in the initializer. Any left over values will be initialized to 0 (all leftover bytes are initialized to 0’s)

Visualization of Structs on Previous Slides

Person 1

age	23
name	“Bob”
gender	‘M’

Person2

23	“Alice”	F’

Designated Initializers (C99)

Designated Initializers (C99)

```
struct{
  int age;
  char name[20];
  char gender;
} person1 = {.name = “Bob”, .age=23, .gender= ‘M’},
  person2 = {.age = 23, .name=“Alice”, .gender=’F’};
```

Designated Initializers (C99)

```
struct{
  int age;
  char name[20];
  char gender;
} person1 = {.name = “Bob”, .age=23, .gender= ‘M’},
  person2 = {.age = 23, .name=“Alice”, .gender=’F’};
```

Note how the order of initialization doesn’t necessary match order of members since the name of the member being initialized is specified

Accessing Struct Members

```
struct{
  int age;
  char name[20];
  char gender;
} person1 = {23, “Bob”, ‘M’},
  person2 = {23, “Alice”, ‘F’};
```

- Use the `.` operator to access struct members:
  - `printf(“age=%d\n”, person1.age);`
  - `person1.age = 55; //change the value of some member`
  - `scanf(“%d”, &person1.age); //read in a value into a member`

Copying Structs

```
struct{
 int age;
 char name[20];
 char gender;
} person1 = {23, “Bob”, ‘M’},
 person2;
```

```
person1 = person2;
```

Copies all values in person2 into the corresponding members in person1. Surprisingly, arrays in structs are “deeply” copied, meaning that each value in the array is copied to the array in the other struct.

demo: struct.c

Declaring Structure Types

Declaring Structure Types

```
struct Person{
  int age;
  char name[20];
  char gender;
};

struct Person p1, p2;
```

Declaring Structure Types

```
struct Person{
  int age;
  char name[20];
  char gender;
};

struct Person p1, p2;
```

this is called using a *structure tag.* When declaring a variable using a defined structure tag, you MUST use the `struct` keyword.

Declaring Structure Types

```
struct Person{
 int age;
 char name[20];
 char gender;
};

struct Person p1, p2;
```

this is called using a *structure tag.* When declaring a variable using a defined structure tag, you MUST use the `struct` keyword.

```
typedef struct{
 int age;
 char name[20];
 char gender;
} Person;

Person p1, p2;
```

Declaring Structure Types

```
struct Person{
  int age;
  char name[20];
  char gender;
};

struct Person p1, p2;
```

this is called using a *structure tag.* When declaring a variable using a defined structure tag, you MUST use the `struct` keyword.

```
typedef struct{
  int age;
  char name[20];
  char gender;
} Person;

Person p1, p2;
```

here, we have created an actual type using `typedef` and do not need to use the `struct` keyword when declaring variables of this type

Structures as Arguments and Parameters

demo: define-struct.c

Nested Structures

Nested Structures

```
struct person_name {
  char first[FIRST_NAME_LEN + 1];
  char middle_initial;
  char last[LAST_NAME_LEN + 1 ];
};

struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

Nested Structures

```
struct person_name {
  char first[FIRST_NAME_LEN + 1];
  char middle_initial;
  char last[LAST_NAME_LEN + 1 ];
};

struct student {
   struct person_name name;
   int id, age;
   char sex;
 } student1, student2;
```

```
…
strcpy(student1.name.first, “Fred”);
```

Nested Structures

```
struct person_name {
  char first[FIRST_NAME_LEN + 1];
  char middle_initial;
  char last[LAST_NAME_LEN + 1 ];
};

struct student {
   struct person_name name;
   int id, age;
   char sex;
 } student1, student2;
```

```
…
strcpy(student1.name.first, “Fred”);
```

useful for grouping related fields

Arrays of Structures

demo: struct-array.c

Arrays of Structures

```
struct person_name {
  char first[FIRST_NAME_LEN + 1];
  char middle_initial;
  char last[LAST_NAME_LEN + 1 ];
};

struct student {
  struct person_name name;
  int id, age;
  char sex;
};

//declares an array of 100 elements,
//each of type struct student
struct student students[100];
…
int i = …;
students[i].id = 40;
students[i].name.middle_initial = 'M';
```

demo: struct-array.c

Database of Information

- Having the ability to create arrays of structures allows us to keep a “database” in our program

- This database can just be used for lookup (can declare the array as const) or can be used to keep track of different entities in our program

- Check the parts database program on p389 of the book.

demo: dictionary.c

Unions

Unions

- Similar to structures, *unions* are a data structure that consists of one or more members, possibly of different types

- However, the difference is that the compiler allocates only enough space for the largest of the members, which overlay each other within this space

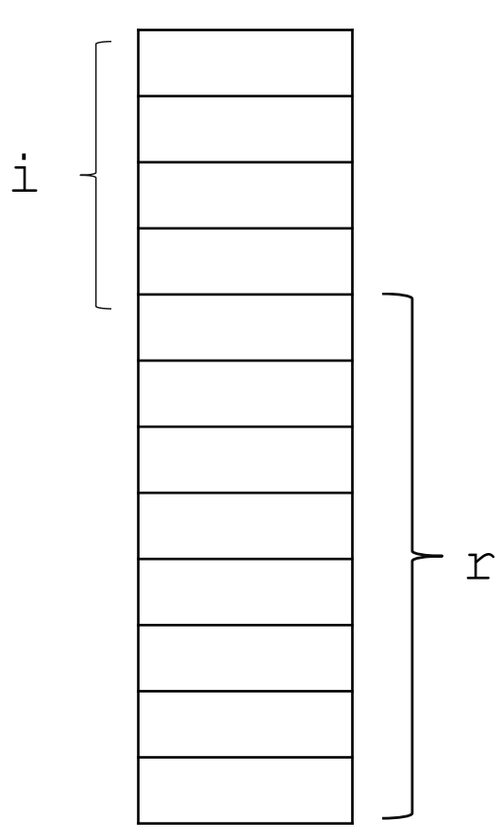
- As a result, assigning a new value to one member alters the value of the other members as well

- Unions are initialized in the same way as structs, and their members are accessed in the same way

Unions

Unions

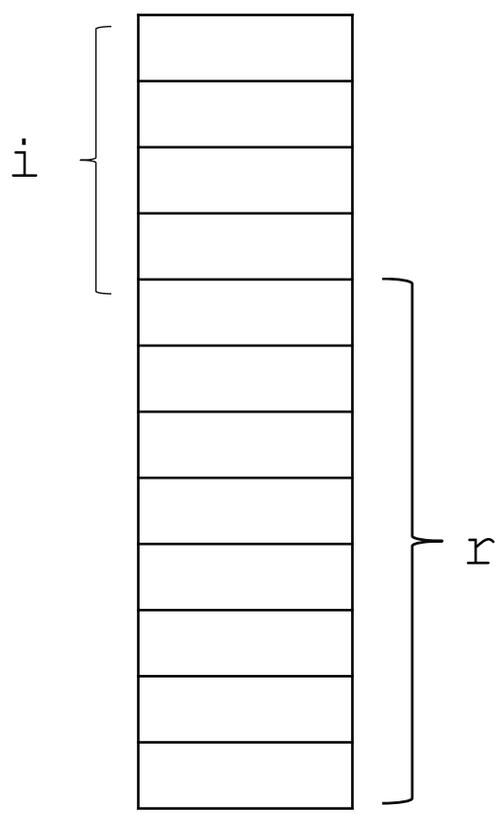
```
struct {
  int i;
  double r;
} d;
```



Unions

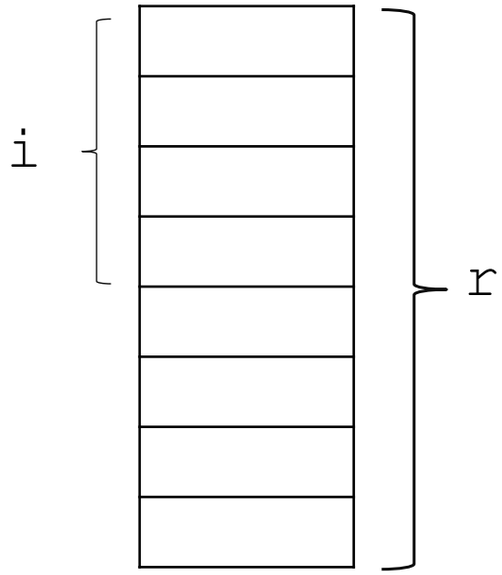
```
struct {
  int i;
  double r;
} d;
```

d




```
union {
  int i;
  double r;
} u;
```

u



When are Unions Useful?

- Unions are often used to save space in structures
- Assume we are designing a data structure to store information about different types of merchandise. Each item has a stock number and a price, as well as other information that depends on the type of the item

demo: member-union.c

When are Unions Useful?

- Unions are often used to save space in structures
- Assume we are designing a data structure to store information about different types of merchandise. Each item has a stock number and a price, as well as other information that depends on the type of the item

```
struct catalog_item{
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN + 1];
  char author[AUTHOR_LEN + 1];
  int num_pages;
  char design[DESIGN_LEN + 1];
  int colors;
  int sizes;
};
```

demo: member-union.c

When are Unions Useful?

- Unions are often used to save space in structures
- Assume we are designing a data structure to store information about different types of merchandise. Each item has a stock number and a price, as well as other information that depends on the type of the item

```
struct catalog_item{
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN + 1];
  char author[AUTHOR_LEN + 1];
  int num_pages;
  char design[DESIGN_LEN + 1];
  int colors;
  int sizes;
};
```

Lots of wasted space!

demo: member-union.c

When are Unions Useful?

- Unions are often used to save space in structures
- Assume we are designing a data structure to store information about different types of merchandise. Each item has a stock number and a price, as well as other information that depends on the type of the item

```
struct catalog_item{
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN + 1];
  char author[AUTHOR_LEN + 1];
  int num_pages;
  char design[DESIGN_LEN + 1];
  int colors;
  int sizes;
};
```

Lots of wasted space!

```
struct catalog_item{
  int stock_number;
  double price;
  int item_type;
  union{
    struct{
      char title[TITLE_LEN + 1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN + 1];
    } mug;
    struct{
      char design[DESIGN_LEN + 1];
      int colors;
      int sizes;
    }shirt;
  } item;
};
```

demo: member-union.c

What are Unions Useful For?

- Unions are also useful to create data structures that contain a mixture of data types

- For example, we might need to create an array whose elements are a mixture of `int` and `double` values.

What are Unions Useful For?

- Unions are also useful to create data structures that contain a mixture of data types

- For example, we might need to create an array whose elements are a mixture of `int` and `double` values.

```
typedef union{
  int i;
  double d;
} Number;

Number number_array[100];
number_array[0].i = 5;
number_array[1].d = 8.395;
```

Adding a “Tag Field” to a Union

Adding a “Tag Field” to a Union

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct{
  int kind;
  union {
    int i;
    double d;
  } value;
} Number;

void print_number(Number n){
  if (n.kind == INT_KIND)
    printf(“%d\n”, n.value.i);
  else
    printf(“%d\n”, n.value.d);
}
```

Enumerations

Enumerations

- Enumerations are useful when we need variables that have a small set of meaningful values. E.g., a variable that stores the suit of a playing card should only have four potential values

- C provides a special type designed specifically for this.

- An *enumerated type* is a type whose values are listed (i.e., "enumerated") by the programer.

Defining Enumeration Types

Defining Enumeration Types

```
enum suit {CLUB, DIAMONDS, HEARTS, SPADES};
typedef enum {club, diamonds, hearts, spa} Suit;
enum suit s1, s2;
Suit s1, s2;
```

Defining Enumeration Types

```
enum suit {CLUB, DIAMONDS, HEARTS, SPADES};
typedef enum {club, diamonds, hearts, spa} Suit;
enum suit s1, s2;
Suit s1, s2;
```

Unlike members of a structure or union, the names of enumeration constants must be different from other identifiers declared in the enclosing scope.

Enumerations: Behind the Scenes

- C treats enumeration variables and constants as integers

- By default, the compiler assignments the integers 0,1,2, .. to the constants in a particular enumeration.

- You can also choose different values:

```
enum suit {CLBS=1, DIAMONDS=30, HEARTS=3, SPADES=5};
```

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant

demo: enum.c