Lecture Notes for Lecture 9 of CS 5001 (Foundations of CS) for the Fall, 201) session at the Northeastern University Silicon Valley Campus.

Defining Types: Enums

Philip Gust,
Clinical Instructor
Department of Computer Science

Lecture 8 Review

- When trying to represent real-world information in a program it is useful to create our own data types.
- Data types are a combination of how data is represented in a program and functions that can operation on the representation.
- Programming languages provide ways to define types including: references sequences, aliases, enumerations, and aggregations.
- We have already studied references in the form of C pointers, and sequences in the form of C arrays

Lecture 8 Review

- Typedefs in C provide a way to alias existing data types for cross-platform portability and to provide more descriptive names
- Application-specific opaque types can be defined that encapsulate their representation outside of functions that operate on them.
- Function identifiers are const pointers to the functions they reference. The type of a function pointer includes the type and number of its parameters and the return type.
- Function pointers are usually typedef'd to reflect what they stand for. This simplifies passing function pointers are arguments to another function.

Enum

- Another way to define a type in a programming language is as a closed set of symbolic values that enumerate a fixed set of choices in the problem being solved.
- Examples include letter grades at an academic institution, or membership categories for an arts organization, or seating areas in a theater.
- Using symbolic names rather than numeric values to represent these choices makes a program easier to design and develop,
- It also ensures that the code is more readable and easier to maintain and extend.
- This lecture will present the facility that the C language provides for defining enumerated data types: *enum*.

Enum

What is an Enum?

- An enum is an enumerated type that specifies a fixed set of values that comprise the type.
- Enum uses identifiers for values that help make code more readable.
- Examples:

```
enum LetterGrade  {A, B, C, D, E };
enum MembershipLevel { members, supporter, friend, patron };
enum SeatingArea {orchestra, loge, mezzanine, balcony };
enum DayOfWeek { sun, mon, tue, wed, thu, fri, sat };
```

Enum

What is an Enum?

- Enum value identifiers must be unique within a program.
- Enum types are in a separate name space so there is no conflict with const, variable, or function types.
- Enum types must be preceded by ‘enum’ as part of the type name in variable or const declarations:

```
enum LetterGrade studentGrade = A;
enum MembershipLevel  memberLevel = patron;
enum SeatingArea ticketSeating = mezzanine;
enum DayOfWeek today = wed;
```

Enum

What is an Enum?

- Enum value identifiers in C are integer constants whose values are known during compilation.
- By default, the first enum value identifier has the value 0, the second the value 1, etc.

```
enum DaysOfWeek { sun, mon, tue, wed, thu, fri, sat };
// sun = 0
// mon = 1
// tue = 2
// wed = 3
// thu = 4
// fri = 5
// sat = 6
```

Enum

What is an Enum?

- While enum value identifiers in C must be unique within a program, the same identifier can appear in multiple enums, but must have the same ordinal value in each.

```
enum DaysOfWeek { sun, mon, tue, wed, thu, fri, sat};  // value of mon is 1
enum Weekdays { mon, tue, wed, thu, fri };    // error – value of mon is not 1
```

- Multiple enum value identifiers can also have the same ordinal value. For example *mon* in DaysOfWeek has the same ordinal value 1 as *feb* in MonthsOfYear.

Enum

What is an Enum?

- Enum value identifiers can be assigned specific values. Here are examples:

```
// weekdays : note second and subsequent values increment from first one
enum Weekdays {mon = 1, tue, wed, thu, fri, sat };

// corresponding numerical grade points
enum LetterGrade  {A = 4, B = 3, C = 2, D = 1 };

//  values returned by a comparison operation
enum Comparison {less = -1, equal = 0, greater - 1};

// minimum donation for membership levels
enum MembershipLevel
    {member = 50, supporter = 100, friend = 500, patron = 1000 };

//  initial of seating area: a char is also a a small integer character code
enum SeatingArea {orchestra = 'O', loge =  'L',  mezzanine = 'M', balcony = 'B' };
```

Enum

What is an Enum?

- The size of an enum value identifier is defined to be “large enough” to hold its ordinal value. In practice, most compilers use at least the size of an int for enum values.

```
enum ManyTypes {
      a=true, b='b', c=-5, d=INT_MIN, e=LONG_MAX, f=LLONG_MAX
};
enum ManyTypes manyTypesVal;
printf("sizeof a: %zu\n", sizeof(a));  // 4
printf("sizeof b: %zu\n", sizeof(b));  // 4
printf("sizeof c: %zu\n", sizeof(c));  // 4
printf("sizeof d: %zu\n", sizeof(d));  // 4
printf("sizeof e: %zu\n", sizeof(e));  // 8 (4)
printf("sizeof f: %zu\n", sizeof(f));   // 8
printf("sizeof manyTypesVal: %zu\n", sizeof(manyTypesVal));  // 8
printf("sizeof enum ManyTypes: %zu\n", sizeof(enum ManyTypes));  // 8
```

Enum

What is an Enum?

- The number of items in an enum is not available in C. Notice that sizeof(*enumtype*) returns the sizeof the largest enum value, not the number of items in the enum.
- There are several strategies for the number of items in an enum, but they must be implemented by the program.
- One is to provide an additional enum value that records it.

```
enum MyEnum { a=10, b=11, c=12, d=13, e=14, f=15, MyEnum_length=6 } ;
```

- Another is to create a separate constant outside the enum.

```
enum MyEnum { a=10, b=11, c=12, d=13, e=14, f=15 } ;
const unsigned MyEnum_length = 6;      // number of items in MyEnum
```

- An advantage of the first is that an enum value is a compile-time constant, while a const variable in C is not.

Enum

Enums and Typedefs

- At first having enum type identifiers in their own *name space*, separate from identifiers for function, variable, const, and typedef, seemed like a good idea to C language developers.
- In practice having a separate name space for enum types had limited value.
- C++, Java, and other programming languages put enum types identifiers in the same namespace other identifiers.
- Most C programmers achieve the same effect by using typedef to create an alias that eliminates the need to use the enum type.

Enum

Enums and Typedefs

- Creating a typedef for an enum allows the typedef type to be used instead of the enum type.

```
enum DayOfWeek { sun, mon, tue, wed, thu, fri, sat };
typedef enum DaysOfWeek DaysOfWeek;  // enum DaysOfWeek is enum type
enum DaysOfWeek yesterday = mon;    // using DaysOfWeek enum type
DaysOfWeek today = tue;       // using DaysOfWeek typedef
```

- The enum and typedef can be combined into one declaration.

```
// combine enum and typedef definitions into one declaration
typedef  enum DaysOfWeek { sun = 0, mon, tue, wed, thu, fri, sat } DayOfWeek;
```

- The enum type name can be left off if is is not needed. This is known as an "*anonymous enum.*"

```
// omit enum type name and only refer to typedef type name
typedef  enum { sun, mon, tue, wed, thu, fri, sat } DayOfWeek;
```

Enum

Example: Using Enum to Define a Boolean Type

- Another way to define type *bool* before it was introduced in C99, was to use a bool enum with false (0) and true (1).

```
// represent bool using an 8-bit unsigned int
typedef enum { false, true } bool;

/**
  * Return bool value true if input value is even, false otherwise
  * @param val the input value
  * @return true if even, false otherwise
  */
bool isEven(int val) {
   return (val % 2 ==0) ? true: false;
}
```

Enum

Using Enums as Loop and Array Indexes

- Because enum ordinal values are integers, they can be be used as loop and array indexes.

```
// define enum symbolic values for weekdays (sun=0)
typedef enum {mon=1, tue, wed, thu, fri } WeekDays;

// array for names of weekdays (unused entry at index 0)
const char *dayName[] = {”", "Mon", "Tue", "Wed", "Thu", "Fri" };

// print numeric and symbolic values of week days (1, 2, 3, 4, 5)
for (WeekDays day = mon; day <= fri; day++) {
    printf(”day: %d name: %s\n”, day, dayName[day]);
}
```

Enum

Using Enums in Switch Statement

- Enum ordinal values are literal integer constants, so they can be used as switch values and labels.

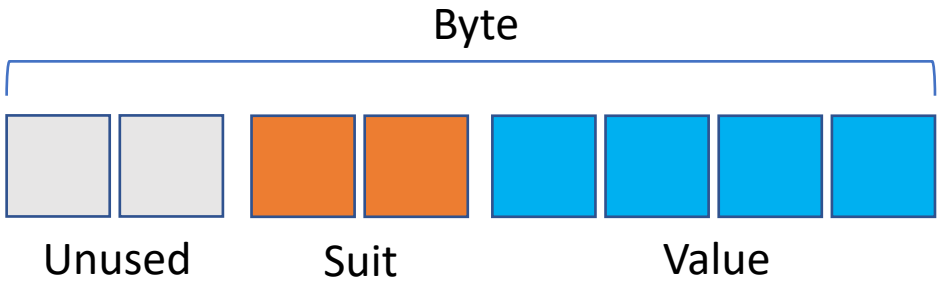
```
typedef enum { X, O, NONE } Marker;       // typedef enum for TTT markers
typedef Marker TTTBoard[3][3] ;           // typedef for TTT board

/** Return character corresponding to marker to print board.
  * @param marker the board marker
  * @return character corresponding to marker
  */
char markerToChar(Marker marker) {
    switch(marker) {                  // dispatch on value of marker
          case X:    return 'X';            // 'display X marker as 'X'
          case O:    return 'O';            // display O marker as 'O'
          case NONE:  return ' ';           // display no marker as space char ' '
          default: return '? ';             // display unknown marker as '?'
    }
}
```

Enum

Example: Implementing a Playing Card

- Cards in a deck of playing cards include four suits: hearts, diamonds, clubs, and spades, and thirteen values: ace, 2, 3, 4, 5, 6, 7, 8, 10, jack, queen, and king (13 values).
- One way to represent a card is as a uint8_t that combines the suit and value, shifting the suit left by 4 and ORing the value:



Enum

Example: Implementing a Playing Card

```
#include <stdio.h>
#include <stdlib.h>

/** Playing card  encodes the suit and value */
typedef uint8_t Card;

/** Card suits */
typedef enum {
        hearts, diamonds, clubs, spades
} CardSuit;

/** Card values */
typedef enum {
        ace = 1, two, three, four, five, six, seven, eight, nine, ten, jack, queen, king
} CardValue;
```

Enum

Example: Implementing a Playing Card

```
/** Names of suits */
const char* cardSuit[] = { "hearts", "diamonds", "clubs", "spades" };

/** Names of cards */
const char* cardValue[] = {
        "", "ace", "two", "three", "four", "five", "six", "seven",
        "eight", "nine", "ten", "jack", "queen", "king"
};

/** Return a card with a suit and value
 * @param suit the suit
 * @param value the value
 * @return the card
 */
Card makeCard(CardSuit suit, CardValue value) {
        return suit << 4 | value;
}
```

Enum

Example: Implementing a Playing Card

```
/**
 * Get the value of a card.
 * @param card the card
 * @return the card value
 */
CardValue getCardValue(Card card) {
        return card & 0xF;
}

/** Get the suit of the card.
 * @param card the card
 * @return the card suit
 */
CardSuit getCardSuit(Card card) {
        return  card >> 4;
}
```

Enum

Example: Implementing a Playing Card

```
/**
 * Get string representation of card (e.g. "ace of diamonds")
 * @param card the card
 * @param strbuf the result string array
 * @return pointer to the result string array
 */
char *cardToString(Card card, char *strbuf) {
        int value = getCardValue(card);
        int suit = getCardSuit(card);
        sprintf(strbuf, "%s of %s", cardValue[value], cardSuit[suit]);
        return strbuf;
}
```

Enum

Example: Implementing a Playing Card

```
/** Test card functions */
int main(void) {
        char cardbuf[20];

        for (CardSuit suit = hearts; suit <= spades; suit++) {
                for (CardValue value = ace; value <= king; value++) {
                        Card card = makeCard(suit, value);
                        printf("%s\n", cardToString(card, cardbuf));
                }
                printf("\n");
        }
}
```