The Rest of C

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Enumerated types
- Unions
- volatile
- restrict

Enumerated Types

- Often, we need types with small, fixed sets of values
  - e.g., Color with values RED, GREEN and BLUE
  - e.g., Direction with values UP, DOWN, LEFT, RIGHT
- We have a way to handle this … but it's not a great way
- What's wrong with this?
  - These macros don't obey scoping rules
  - `color` could get a value outside this set
  - It's not obvious that color has a restricted type

```
#define RED 0
#define GREEN 1
#define BLUE 2
```

```
int color = BLUE;
```

Enumerated Types

- C lets us define new types, with any set of values we want

```
enum ColorEnum { red, green, blue, orange };
```

- Then, we can use this type, kind of like a structure tag

```
enum ColorEnum myColor = green;
```

- You can compare enum values for equality/inequality

```
enum ColorEnum yourColor = red;
if ( myColor == YourColor ) {
   …;
```

Enumerated Type Names

- Or, we could use typedef to get a shorter name:

```
typedef enum { up, down, left, right } Direction;
```

```
Direction myDirection = right;
```

- Internally, enumerated values are stored as small integers
  - They start at zero and count up for each value.

```
printf( "%d\n", myDirection );
```

This will print 3.

  - And, you can assign between enum and int values.

```
int val = myDirection;
```

Enumerated Type Names

- You can choose numeric values for each type
  - Specify integer values for any enumeration constant

```
enum MoodEnum { happy = 2, afraid, bored = 0,
                blue, glad  };
```

So, happy and glad are the same thing … makes sense.

- You can even do math on enum values … but be careful.

```
enum MoodEnum myMood = happy;
myMood++;       // I'm feeling afraid.
myMood++;       // Now, I don't know how I feel.
```

Enumerated Types

- Enumeration values have scope, like a type or a variable.
  - So, you can't have identical values in multiple enumerations.

```
enum ColorEnum { red, green, blue, orange };
enum MoodEnum { happy, bored, afraid, blue };
```

  - But, you can if they are at different scopes:

```
enum ColorEnum { red, green, blue, orange };
{
  enum MoodEnum { happy, bored, afraid, blue };
  …;
```

Reminder Compile-Time Constants

- C Requires some program elements to be compile-time constants
  - Size of a static array, case values in a switch, size of an array inside a struct.
- For example, the sizes of the following arrays.

```
typedef struct {
  char title[ 100 ];
  char author[ 100 ];
  long isbn;
} Book;
```

Reminder Compile-Time Constants

- This is a good example of a magic number we'd like to avoid.
  - We can get help from the preprocessor:

```
#define SIZE 100

typedef struct {
  char title[ SIZE ];
  char author[ SIZE ];
  long isbn;
} Book;
```

- But, this isn't ideal, now nobody else can use SIZE

The enum Hack

- Enum values are compile-time integer constants.
- People have found a clever way to use this feature.
  - This is called the *enum hack*

```
enum { SIZE = 100 };
int list[ SIZE ];
```

  - It respects scope (which we don't get with #define)

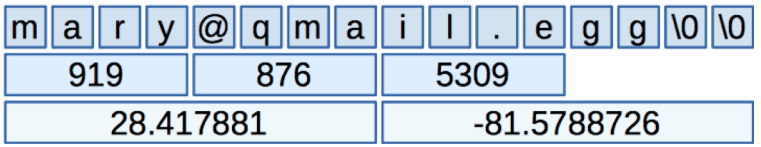
```
typedef struct {
  enum { SIZE = 100 };

  char title[ SIZE ];
  char author[ SIZE ];
  long isbn;
} Book;
```

Unions

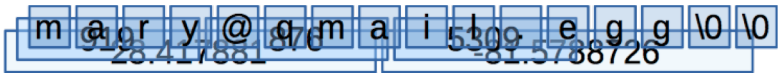
- A struct organizes a collection of fields, one after another in memory

```
struct ContactInfo {
  char email[ 25 ];
  short phone[ 3 ];
  double coords[ 2 ];
};
```



- A union organizes a collection of fields, all occupying the same memory

```
union ContactInfo {
  char email[ 25 ];
  short phone[ 3 ];
  double coords[ 2 ];
};
```



Unions

- So, we can use any of the fields
  - But just one at a time.

```
union ContactInfo bill;

bill.phone[ 0 ] = 919;
bill.phone[ 1 ] = 867;
bill.phone[ 2 ] = 5309;

union ContactInfo mary = { .email = "mary@qmail.egg" };

printf( "Phone: %03d-%03d-%04d\n", bill.phone[ 0 ],
        bill.phone[ 1 ], bill.phone[ 2 ] );

printf( "email: %s\n", mary.email );

printf( "Phone: %03d-%03d-%04d\n", mary.phone[ 0 ],
        mary.phone[ 1 ], mary.phone[ 2 ] );
```

Good, bill stores a phone number.

And mary holds an email.

But, I bet this will be ugly.

Unions

- Why would you use this? To save memory.
  - Normally, you'd use a union as part of a larger structure.
  - The union would let you re-use the memory for different things.
  - Different instances would use different fields of the union.

```
struct Contact {
  char name[ 20 ];

  enum { email, phone, coords } type;

  union ContactInfo info;
};

struct Contact contacts[] = {
  { "Mary", email, { .email = "mary@qmail.egg" } },
  …
};
```

I can just store one type of contact information.

So, I take less total memory.

Unions

- Unions don't automatically remember which field you're using
  - You have to keep up with that yourself.

```
struct Contact {
  char name[ 20 ];

  enum { email, phone, coords } type;

  union ContactInfo info;
};

struct Contact contacts[] = {
  { "Mary", email, { .email = "mary@qmail.egg" } },
  …
};
```

That's what I'm for.
I remember what field you're using in the union.

If you're storing an email, remember that.

Unions

- Whenever you use the union, you probably need to check what you stored there.

```
struct Contact contacts[] = {
  { "Mary", email, { .email = "mary@qmail.egg" } },
  { "Bill", phone, { .phone = { 919, 867, 5309 } } },
  …
};

for ( int i = 0; i < sizeof(contacts)/sizeof(contacts[0]); i++ ) {
  if ( contacts[ i ].type == phone )
    printf("phone: %03d-%03d-%04d\n", contacts[i].info.phone[0],
           contacts[i].info.phone[1], contacts[i].info.phone[2] );
  else if ( contacts[ i ].type == email )
    …
```

What field am I using?

The Usual

- Of course, you can use typedef to give your union its own type name:

```
typedef union {
  char email[ 25 ];
  short phone[ 3 ];
  double coords[ 2 ];
} ContactInfo;

ContactInfo bill;
```

See, now you can make instances without the union keyword.

- But, this usually isn't necessary.
- More often, union types don't even need names.

The Usual

- We usually use unions as part of a struct.
- Inside the struct, we need a field that's a union
  - but we may not need a name for the union type itself.

```
struct Contact {
  char name[ 20 ];

  enum { email, phone, coords } type;

  union {
    char email[ 25 ];
    short phone[ 3 ];
    double coords[ 2 ];
  } info;
};
```

This type has no name.

Want to use it?
Just use this field.

Not using Unions

- There's nothing wrong with unions .. in C
- Java would never let us do something like this.
  - Java runs a tight ship
  - It can't let us spontaneously change our minds about what's stored in a region of memory.
- But, having objects that can vary can be useful
- And, we do this in Java … all the time … how?
  - Inheritance
  - We can have a superclass with common fields and behavior
  - And subclasses with specialized fields and behavior
- C++ will let us do the same thing

Volatile

- You can mark some variables as volatile.

```
volatile int x = 0;
```

- What does it mean?
  - It's a hint to the compiler, sort of the opposite of register.
  - "Go to memory every time to get this value."
  - "Don't save it in a CPU register."
  - Useful for Operating System or parallel programming (you'll see it again in OS class)

Restrict

- Restrict is kind of like const.
- It says “no other pointer in this block of code will get you to the same memory location.”
- A hint during compiler optimization.

```
void *memcpy( void *restrict dst,
              const void *restrict src,
              size_t n );
```

We don’t overlap.

```
void *memmove( void *dst,
               const void *src,
               size_t len );
```

But, we might.