ECE 177 – Programming I: From C Foundations to Hardware Interaction Lecture 29

Vince Weaver

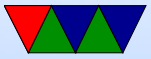
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

15 April 2026

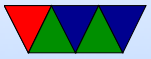
Announcements

- Hopefully Lab#9 going OK
- Let me know if you are having VS code problems
- Still grading second midterm



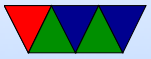
Collision Detection Aside

- I gave you some code that does "barely good enough" detection
- It turns out good collision detection is surprisingly complicated
- In this case when it's two rectangles and you know one is always bigger than the other you can more or less get away with 4 compares like we do here
- Ideally you'd detect X and Y separately and bounce in the x direction too, or bounce both x and y if you run



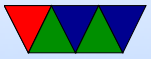
into a corner

- Trouble if moving too fast and clip through things or miss all together, in that case need to check all points not just edges
- Minkowski difference
  `https://spader.zone/minkowski/`



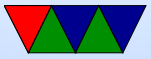
Pointer Comparison

- You can check if equal or not-equal
- You can also check greater than or less than, though again ideally it would be with pointers inside the same object, and also again why would you want to do this



Converting Pointer to Integer

- Generally a “long” can hold a pointer
- You can cast between them
- Why would you want to?
- Generally don’t
- One reason is if you are trying to print it, `printf()` internally will do that if you use `printf("%p",pointer);`



const pointers

- You can declare a constant pointer: the pointer will never change (but the data being pointed to can)

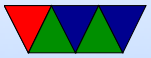
```
char *const ptr="Hello";
```

- You can declare a pointer to a const variable: the pointer can change (but the data being pointed to can't)

```
const char *ptr="Hello";
```

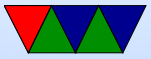
- Can you have a const pointer to a const variable? Yes. Why?

```
const char *const ptr="Hello";
```

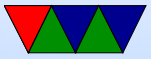


const strings

- You might see this most with strings.
- C++ in particular likes using const pointers for strings passed into functions
- Why? It indicates that the function isn't going to modify the string
- Also if you have const strings the compiler can put the string data in a read-only part of memory which can avoid bugs where strings were unintentionally written
- Manpage has printf declared as

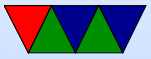


```
printf(const char *restrict format,
```



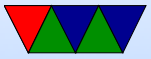
Strict Aliasing

- C can have issues with all these pointers
- A compiler can sometimes not make certain optimizations because it can't guarantee there isn't a pointer floating around that will let some other code modify a variable out from under it
- gcc `-no-strict-aliasing` option



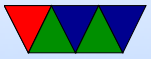
restrict keyword

- C99 added a restrict keyword
- It tells the compiler there's only one pointer to a memory region
- This lets the compiler do a better job of optimizing code

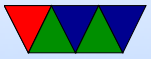


aside – register keyword

- Generally it's expected variables live in memory
- Access to memory is slow though
- Compiler (with optimizations enabled) can bring a value into a register and operate on it there and only write back when done
- In the old days you could use the `register` keyword to tell the compiler this was OK `register int x;`
- It also implied you'd never take the pointer of this variable



- These days the compiler can figure stuff out automatically so you don't see `register` used a lot

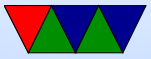


volatile

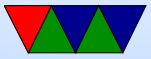
- A C compiler might cache a value in a register and not re-read it each time if it thinks nothing can change it.

```
x=5;
....
....
y=x;
```

- Can it assume y will always be 5 here?
- What if there was a pointer to x and it was changed?
- You can force the C compiler to read a value from memory (or not optimize) with the volatile keyword



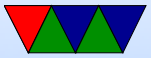
- `volatile int x;`



volatile pointers

- Volatile pointer to non-volatile data?
  Not really that useful
- Volatile pointer to volatile data?

- Useful because if you use a regular pointer to point to a volatile variable, the compiler doesn't know it's volatile anymore if you access it through a normal pointer
- `volatile int *ptr;`

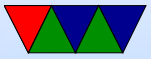


volatile – using with hardware access

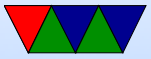
- One case this is used is when doing low-level hardware

```
while(sio->gpio18==0) {
```

- As far as the compiler can tell, that value will never change, but if it's hardware memory-mapped I/O address it actually can
- Often on embedded systems like this there's a hack where you declare it volatile and that will work
- The more "proper" way to do things is to have inline assembly and memory barriers but that's a lot more work

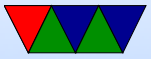


- Similar issues can come up if you are on multi-core processors and have shared memory. Again using volatile would be considered a hack



typedef

- Were you disappointed by how few native types C has?
- You can use structs to make something more interesting
- But isn't it annoying to have to have the struct keyword everywhere? `struct ball_stats ball;`



typedef – syntax

- Basically lets you assign a new name to an existing type.

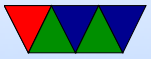
```
typedef int color;

color a=0;
```

- Often used for structs

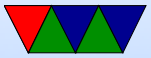
```
typedef struct ball_stats ball_t;

ball_t ball;
```



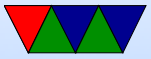
typedef – why

- Make code easier to follow?
- You would think maybe you can get errors if you assign from one type to another, but C doesn't do that



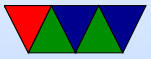
typedef – against

- Some people don't like using typedefs
- Instead of making this easier to follow, can just make it more confusing as you have to look up what the type actually is
- It doesn't really save a lot of space either
- I personally try to avoid them



unions

- Similar to a structure
- But what if you want some of the fields to *overlap* in memory?
- Why would you *ever* want to do that? Well, RAM was expensive in the 1970s
- So you can specify multiple types in a union, but the memory allocated is only that of the biggest type, and it can only hold one of the values at a time



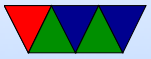
union syntax

```
union foo {
     int x;
     char c;
     double b;
     char arr[5];
};

union foo our_union;

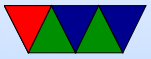
our_union.x=5;
printf("%d %lf\n",our_union.x,our_union.b);

our_union.b=3.14;
```



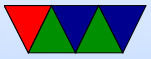
```
printf("%d %lf\n",our_union.x,our_union.b);
```

- Has the size of the biggest element (the double in this case, 8 bytes)
- You can access any of the fields, but they share memory so they overlap
- If you write a double then read as an int you just get the raw bit pattern of the bottom 4 bytes treated as an int



unions – why?

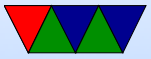
- To save memory
- You might have a generic structure, with first element type
  if type is 1, coord you want there to be int x,y if type is 2 it's color you want color c it's never possible to be both coord and color at same time so would be a waste to allocate space for all just in case



type punning

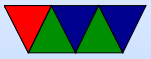
- Trying to convert raw bit pattern from one type to another
- Say you have a floating point number, but want to raw bits for some reason (want to mask it to print?)
- Traditionally people might use unions for this even though questionable
- Can also try to use pointers

```
float f;
int i=f;  // doesn't work, C converts type
i=*(int *)&f;
```



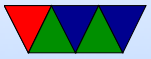
- I think official way is to instead use memcpy()

```
memcpy(&i,&f,4);
```



bitfields

- Using an integer to only hold small values (1 bit, 3 bit, etc) is wasteful
- Wouldn't it be great if you could have an integer and split it up into different chunks?
- You can do this manually but it involves lots of bit-shifting, masking or-ing, etc
- Wouldn't it be great to make the compiler do it for you!
- Unfortunately not very portable so often discouraged



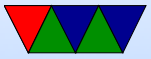
syntax

```
struct car_type {
  unsigned type:4;
  unsigned color:3;
  unsigned electric:1;
} car;

car.electric=1;
printf("Car is electric: %d\n",car.electric);

// instead of
result=(car>>4)&1;
```

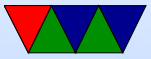
- sizeof shows car is 4 bytes (an int)
- can't run sizeof on bitfield



- What happens if you try to put a value too big in?

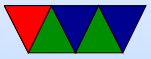
```
car.electic=100;
```

- gcc at least warns you



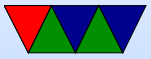
bitfield downsides

- cannot have arrays of bitfields
- compiler might add padding (non-portable)
  makes it unsuitable for binary formats or accessing hardware
- can't take a pointer to one



enums

- Enumerated Data Type
- Common in other languages, let you create a type that has only a certain number of allowed values

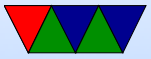


alternative to enums

- Use defines, but no safety and you can accidentally assign to the wrong type

```
#define COLOR_RED    0
#define COLOR_BLUE   1
#define COLOR_GREEN  2
#define COLOR_YELLOW 3

int color=COLOR_BLUE;
int xsize=COLOR_YELLOW;
```



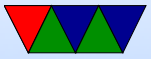
enum – syntax

- ```
  enum color {RED, BLUE, GREEN, YELLOW};

  enum color foreground=RED;
  ```
- You can use typedef so you don't need to say enum each time
- Underneath it is just secretly an integer so you can do integer math even if it doesn't make sense

  ```
  enum color bg=RED+BLUE;
  ```
- You can force the values in the enum

  ```
  enum color {RED, BLUE, GREEN, YELLOW=25};
  ```


  ```

enum – type safety

- Again there is none

```
enum color {RED, BLUE, GREEN, YELLOW};
enum color fg = RED;
enum ice_cream {VANILLA, CHOCOLATE, STRAWBERRY};
enum ice_cream flavor;
flavor = GREEN;
```

- Also issues with scope, even though RED declared in an enum, it means you can't use as a var name or even use in a separate enum