ECE 177 – Programming I: From C Foundations to Hardware Interaction Lecture 4

Vince Weaver

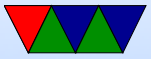
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

30 January 2026

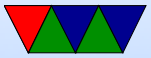
Announcements

- Will take Attendance
- Homeworks, still working on getting that set up
- If you are mailing me about missing class, please mention it's ECE177 you are missing and ideally the exact dates.



Lab Update

- Sorry going a bit poorly. As a computer engineer I always feel responsible when a computer is not working
- Getting things going from directions and having them work out is sadly not uncommon as a computer engineer. Please don't let it scare you away!
- Possibly some of the troubles on Windows are if OneDrive enabled, are working on updated documentation

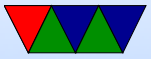


Numbering Systems – Base 10

- Most people use base 10: digits 0..9
- Place Value, $31,415.92$:

3	1	4	1	5	.	9	2
10^4	10^3	10^2	10^1	10^0	.	10^{-1}	10^{-2}

- This is arbirary, can do with any base

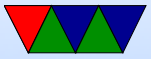


Binary Numbers – Base 2

- Computers inside use base 2: digits 0,1
- 1010 1100

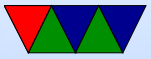
1	0	1	0	1	1	0	0	.	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}
128	64	32	16	8	4	2	1	.	

- Powers of two (do you have them memorized)
- In C `int i=0b10101100;` nonstandard until C23
  gcc supported earlier?
- The numbers get a bit long for 32-bit and 64-bit



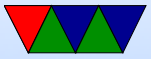
Octal Numbers – Base 8

- Base 8: digits 0..7
- C supports this because early UNIX systems used it, still see in things like permissions chmod 0777
- If you group binary bits by 3 then easily convert to octal `001.010.100 = 0124`
- In C indicate with leading zero
- NOTE! This can cause bugs! People think 0123 is decimal and you can ignore leading 0, but no, it's actually octal 1*64+2*8+3=83



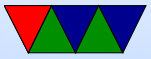
Hexadecimal Numbers – Base 16

- Base 16, digits 0..9,A..F
- Very common way to write large binary numbers
- If you group by 4 then easy to convert binary to hex 1010.1000.0001.1111 = 0xA81F
- Can spell things, 0xdeadbeef, 0xcafebabe, etc
- In C use leading 0x, `int i=0xfeb13;`
  (Aside, old 8-bit systems use $ or h, ie. $1234 or 1234h)



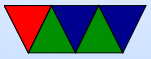
Converting Binary to Decimal

- Things like printf() might do this
- Take number, divide by 10
  Remainder is digit, quotient divide again
  123 / 10 = 10R3, 10/10=1R2 1/10 = 0R1
  Get results right to left



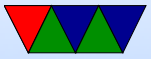
Converting Decimal to Binary

- Just eyeball it and grab the values
- Method: Divide by 2, Remainder. 10 /2 = 5R0, 5/2=2R1 2/2=1R0 1/2=0R1 = 1010



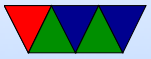
Hex to Binary and Binary to Hex

- Much easier, which is why we use Hex
- Just convert 4 bits at a time
- When you learn more digital logic, divide by 16/remainder (mod by 16) are just shifts and ands



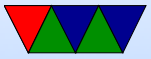
Signed Numbers

- We talked about "signed" integers. How does that actually work? And why is there a distinction?
- On modern computers, when you have a binary number in memory your CPU has no idea if a number is signed. It just sees an 8-bit integer or whatever
- It's a software convention to treat this pattern as signed or unsigned

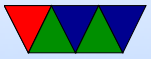


One's Complement

- This is the most straightforward way to do things, but essentially no modern computers use it
- To treat a number as negative just set the high bit to 1.
- `0000.0001 (0x01)` is positive 1
  `1000.0001 (0x81)` is negative 1
- For 8-bit number, 0..127 same as normal, but you can't have values higher than that. unsigned 128..255 become -0,-1,..-127
- Hardware has to special case adds/subtracts



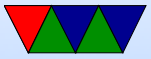
- Weird corner case: there are two zeros (one positive, one negative)
- The C spec in theory allows machines using one's complement for historical reasons



Two's Complement

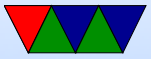
- To convert a number to be negative, flip each bit (0 to 1, 1 to 0) and add 1
- The top bit will be 1 if negative
- What is -1 in two's complement? Take 1 and negate it

```
                0 0 0 0   0 0 0 1     (0x01)
flip
                1 1 1 1   1 1 1 0     (0xfe)
now add 1
                1 1 1 1   1 1 1 1     (0xff)
```



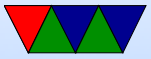
`(ignore carry out of top bit if happens)`

- So -1 is 0xff in two's complement
- Addition/subtraction work same for signed and unsigned numbers
- signed: $1 + \text{-}1 = 0$ (0x01+0xff = 0x00, overflow ignored)
- unsigned: $1 + 255 = 0$ (unsigned wrap around to 0 when overflow)
- Corner case: you can represent -128 (0x80) but the highest number is 127 (0x7f)



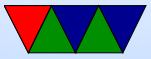
Types – Reminder

- char, short, long, long long, float, double
- What can you assign to these?



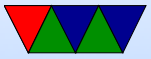
Separating Statements

- We showed last class you can run together your code all on one line, C has no rules about it
- How does the compiler keep the separate statements apart?
- You have to have a semicolon separating one from the next
- `int i=5; char j=3;`



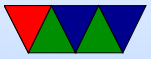
Assigning Values to Variables

- After declaring a variable we can assign a value to it (that's the name, its value can vary)
- Sort of showed this earlier
- Use equals sign to assign a value
- `int i=5;`
- Be careful, if you come from a language where = means check if equal and has something else that means assign (:= pascal) this can trip you up and even lead to serious bugs. C compiler better about warning you



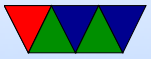
Assigning Values from other Variables

- `int x=5; int y; y=x;` After this what value does y have? (5, same as X)



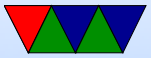
Assigning Values from other Types

- `int i=5; char c;`
- Can you do `c=i;`?
- In a strongly typed language this would give you an error
- C will happily do this for you
- What happens if the value you are trying to assign won't fit?



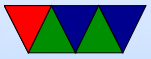
Converting between Types / Promotion

- If types are the same, no problem
- When they are different, lower type converted to higher type
- C always promotes lower value up
- Nothing is demoted until end when the value is stored
- If doesn't fit, truncates (it doesn't round)
- This is often what you want, but not always.



Implicit Promotion Example

- ```
  int a=5;
  char c,b=3;
  c=a+b;
  ```
- When adding a+b, the char b is converted up to an integer size to do the math
- When done and assigning back to c, the result is truncated to fit into a char

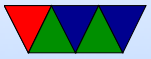

  ```

Implicit Promotion Floating Point

- A similar thing happens when converting from int to float types
- Note again: no rounding happens. So if

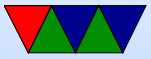
```
float f=2.8;
int i;
i=f;
```

- i wil get the value 2, not 3



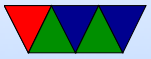
Constants or Literals

- Besides variables you'll notice we have been using constants
  - 1 is an integer
  - 1.0 is a floating point value
  - C will (possibly) pick the smallest integer size that a constant will fit in



Forcing Constant Size

- Integer
  - 1u or 1U = unsigned int
  - 1ul or 1UL = unsigned long
  - 1l or 1L = signed long
- Floating Point
  - 1.0f or 1.0F = 32-bit float
  - 1.0l or 1.0L = 64-bit double



When would you need to specify?

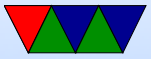
- Usually when mixing different data sizes
- We said promotes up. So if you do something like

```
signed char y;
int x = 127 * y;
```

  and y is 10 then this might do math at 8 bit, overflow, then upgrade to int after, whereas

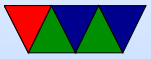
```
signed char y;
int x=127L * y;
```

  this would force the math at 32-bits



Aside: Sign Extension

- We said top bit 1 means negative and also modern CPUs can not worry about sign, but there is one case it does matter, when changing types
- But what if 8-bit negative 1 (0xff) but we assign it to a 32 bit value
- Needs to be 0xffffffff
- If positive number, "extend" all bits from 8 to 31 as 0
- If negative number, "extend" all bits from 8 to 31 as 1
- Hardware can do this by just duping the high bit



Character Constants

- Use single quotes
- `char c='A';`
- Sets to ASCII equivalent
- Special characters can be escaped
  - `'\t'` is tab
  - `'\n'` is linefeed
  - `'\r'` is carriage return
  - `'\b'` is backspace
  - `'\a'` is alert (bell/beep)