C data types

int, long

double
(rarely "float")

char
→ just small ints.

"literals"
38
38L
0x2a
033
Floating-point literals:
0.3
3.0
6.02e23

printf(format, args ...)

printf key letters:

- d: integer, format in base ten ("decimal")
- f: double, traditional formatting
- g: double, flexible formatting
- e: double, scientific notation

Modifiers (between '`%`' and the key letter):

- a number is a field width
- '`.`' and a number is a "precision"
  - Example: `printf("%6.3f", 2.8)` yields `_2.800`
- `0` means pad with zeroes to field width
- `l` for "long", e.g. `%ld` to format a long int in decimal

Example of '0':

```
int dollars = 2;
int cents = 3;  /* $2.03 */
printf(" ?? what goes here ??  ", dollars, cents);
```

```
"$%d.%d"    → 2.3
"%d.%2d"    → 2. 3
"%d.%02d"   → 2.03
```

The '0' is not part of the number; it's a flag!

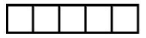
In C, most things are ints. (exaggeration)

- ints are used as booleans
  - relational ops yield 1 or 0
  - conditional contexts test non-zero versus zero
- char literals are ints!
  - (as opposed to strings, later)
  - 'a'
  - '\033'
  - '\n'

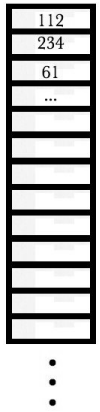
Arrays

Arrays in C aren't objects.

If an int looks like this: 

then an array of five ints looks like this: 

Memory is a vast array of bytes:



Addresses:

A particular int variable might occupy memory locations 1234, 1235, 1236, and 1237.

```
int a[10];
```

Suppose ints are four bytes, and 'a' starts at location 1234.

Then `a[i]` starts at location `1234 + 4 * i`

"pointer": high-level version of an "address"
→ has type information

```
int i;
int *p;
```
→ declare `p` to be of type pointer-to-int

```
i = 3;
p = &i;
```
→ assign `p` to point to `i`

```
printf("%d\n", *p);
```
→ "dereference" — follow a pointer

```
i = 4;
printf("%d\n", *p);
*p = 5;
```

scanf

```
int x;
printf("Enter x: ");
scanf("%d", &x);
```

Why a pointer?

scanf key letters:

- `d`: decimal integer
- `f`: float, any formatting
- `lf`: double, any formatting

Be careful about the type difference between `f` and `lf`! It's different from printf, to which it's impossible to pass a 'float' because of the default promotions.

"pointer arithmetic"

If `p` is a pointer to int,
`p + 3` yields the address which is three ints later

So if `p` contains the address of `a[0]`, `p+i` is the address of `a[i]`.

For two expressions `x` and `y`,
`x[y]` is defined as `*((x) + (y))`

An array name in an expression context decays into a pointer to the zeroth element.

`a[3]`

- the 'a' becomes a pointer to the zeroth element (pointer-to-int)
- add three to get a pointer to element number 3
- dereference

Array parameters

```
#include <stdio.h>

int main()
{
    int a[10];
    extern void setsquares(int *p);
    setsquares(a);
    printf("three squared is %d\n", a[3]);
    return(0);
}

void setsquares(int *p)
{
    int i;
    for (i = 0; i < 10; i++)
        p[i] = i * i;
}
```

Special weird rule for formal parameters only:
You can write `void setsquares(int a[])` !

I strongly recommend against this.
1) It is still a pointer!
2) This weird conversion rule is specific to formal parameter lists!

```
#include <stdio.h>

int main()
{
    int a[10];
    extern void setsquares(int *p, int size);
    setsquares(a, 10);
    printf("three squared is %d\n", a[3]);
    return(0);
}

void setsquares(int *p, int size)
{
    int i;
    for (i = 0; i < size; i++)
        p[i] = i * i;
}
```

Null pointers in C

- The concept of a null (or "nil") pointer value:
  - special signal value
  - guaranteed not to compare equal to a pointer to any actual value of that type
- How we get it in C is weird: convert a constant zero to pointer type and it becomes a null pointer of that type.
- Obviously, history in sloppiness...
- Rather than using 0 for this purpose, we normally use NULL, which is just defined as 0 in stdio.h.