C data types and declarations

(Reek, Ch. 3)
Four basic data types

- **Integer**: char, short int, int, long int, enum
- **Floating-point**: float, double, long double
- **Pointer**
- **Aggregate**: struct, union
  - Reek categorizes arrays as “aggregate” types – fair enough, but as we’ve seen, arrays also have a lot in common with pointers
- Integer and floating-point types are atomic, but pointers and aggregate types combine with other types, to form a virtually limitless variety of types
Characters are of integer type

- From a C perspective, a character is indistinguishable from its numeric ASCII value – the only difference is in how it’s displayed.

- Ex: converting a character digit to its numeric value
  - The value of '2' is not 2 – it’s 50
  - To convert, subtract the ASCII value of '0' (which is 48)

```c
char digit, digit_num_value;
...
digit_num_value = digit - '0';
```

Behaviorally, this is identical to
```
digit - 48
```
Why is
```
digit - '0'
``` preferable?
Integer values play the role of “Booleans”

- There is no “Boolean” type
  - Relational operators (==, <, etc.) return either 0 or 1
  - Boolean operators (&&, ||, etc.) return either 0 or 1, and take any int values as operands
- How to interpret an arbitrary int as a Boolean value:
  - 0 → false
  - Any other value → true
The infamous = blunder

- Easy to confuse equality with assignment
  - In C, the test expression of an if statement can be any int expression — including an assignment expression

```c
if (y = 0)
    printf("Sorry, can't divide by zero.\n");
else
    result = x / y;
```

- The compiler will not catch this bug!
The less infamous “relational chain” blunder

- Using relational operators in a “chain” doesn’t work
- Ex: “age is between 5 and 13”

A correct solution: $5 \leq \text{age} \land \text{age} \leq 13$

evaluate $5 \leq \text{age}$
result is either 0 or 1

Next, evaluate either
$0 \leq 13$
or
$1 \leq 13$
result is always 1

A correct solution: $5 \leq \text{age} \land \text{age} \leq 13$
Enumerated types

- Values are programmer-defined names
- Enumerated types are declared:
  ```c
  enum Jar_Type { CUP=8, PINT=16, QUART=32,
                  HALF_GALLON=64, GALLON=128 };
  ```
  - The name of the type is `enum Jar_Type`, not simply `Jar_Type`.
  - If the programmer does not supply literal values for the names, the default is 0 for the first name, 1 for the second, and so on.

- The ugly truth: `enum` types are just `int`s in disguise!
  - Any `int` value can be assigned to a variable of `enum` type
  - So, don't rely on such variables to remain within the enumerated values
# Ranges of integer types

<table>
<thead>
<tr>
<th>Type</th>
<th>Min value</th>
<th>Max value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>0</td>
<td>UCHAR_MAX (≥ 127)</td>
</tr>
<tr>
<td>signed char</td>
<td>SCHAR_MIN (≤ -127)</td>
<td>SCHAR_MAX (≥ 127)</td>
</tr>
<tr>
<td>unsigned char</td>
<td>0</td>
<td>UCHAR_MAX (≥ 255)</td>
</tr>
<tr>
<td>short int</td>
<td>SHRT_MIN (≤ -32767)</td>
<td>SHRT_MAX (≥ 32767)</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0</td>
<td>USHRT_MAX (≥ 65535)</td>
</tr>
<tr>
<td>int</td>
<td>INT_MIN (≤ -32767)</td>
<td>INT_MAX (≥ 32767)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>INT_MAX (≥ 65535)</td>
</tr>
<tr>
<td>long int</td>
<td>LONG_MIN (≤ -2147483647)</td>
<td>LONG_MAX (≥ 2147483647)</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0</td>
<td>ULONG_MAX (≥ 4294967295)</td>
</tr>
</tbody>
</table>
Ranges of integer types

- Ranges for a given platform can be found at `/usr/include/limits.h`
- `char` can be used for very small integer values
- Plain `char` may be implemented as signed or unsigned on a given platform – safest to “assume nothing” and just use the range `0…127`
- `short int “supposed” to be smaller than `int` — but it depends on the underlying platform
# Ranges of floating-point types

<table>
<thead>
<tr>
<th>Type</th>
<th>Min value</th>
<th>Max value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>FLT_MIN ($\leq -10^{37}$)</td>
<td>FLT_MAX ($\leq -10^{37}$)</td>
</tr>
<tr>
<td>double</td>
<td>DBL_MIN ($\leq -\text{FLT_MIN}$)</td>
<td>DBL_MAX ($\geq \text{FLT_MAX}$)</td>
</tr>
<tr>
<td>long double</td>
<td>LDBL_MIN ($\leq -\text{DBL_MIN}$)</td>
<td>LDBL_MAX ($\geq \text{DBL_MAX}$)</td>
</tr>
</tbody>
</table>

Floating-point literals must contain a decimal point, an exponent, or both.

\[
3.14159 \quad 25. \quad 6.023e23
\]
Danger: precision of floating-point values

- Remember the Patriot story –
  - How much error can your software tolerate?
- Testing for equality between two floating-point values: almost always a bad idea
  - One idea: instead of simply using ==, call an “equality routine” to check whether the two values are within some margin of error.
  - In general, use of floating-point values in safety-critical software should be avoided
Casting: converting one type to another

- The compiler will do a certain amount of type conversion for you:

```
int a = 'A';    /* char literal converted to int */
```

- In some circumstances, you need to explicitly cast an expression as a different type – by putting the desired type name in parentheses before the expression

```
(e.g. (int) 3.14159 will return the int value 3
```
Pointers

- A pointer is nothing more than a memory location.
  - In reality, it’s simply an integer value, that just happens to be interpreted as an address in memory.
  - It may help to visualize it as an arrow “pointing” to a data item.
  - It may help further to think of it as pointing to a data item of a particular type.
A pointer variable is just like any other variable

- It contains a value – in this case, a value interpreted as a memory location.
- Since it’s a variable, its value can change...
- ... and since it occupies some address in memory, there’s no reason why another pointer can’t point to it
Pointers

- Reek uses the metaphor of “street address” vs. “house” to distinguish a pointer (address) from the data it points to
  - OK, but don’t forget that the data at an address may change, possibly quite rapidly
- Maybe a better metaphor: Imagine a parking lot with numbered spaces. Over time, space #135 may have a Ford in it, then a Porsche, then a Yugo,...
  - Here the “pointer” is the space number, and the data is the make of car.
A variable without an initializing expression contains “garbage” until it is assigned a value.

```c
int a;
float f;
char *m, **pm;
/* m is a pointer to char */
/* pm is a pointer to a pointer to char */
```
Variable initialization

```c
int a = 17;
float f = 3.14;
char *m = "dog", **pm = &m;
```

- The string literal "dog" generates a sequence of four characters in memory.
- `m` then points to the first of these characters,
- and `mp` points to `&m`, the address of `m`. 
Array declaration

- Subtle but important point: There are no “array variables” in C. Why not?

```c
int m[4];
```

- The declaration creates a sequence of four spaces for chars.
- The array name `m` refers to a `constant` pointer – *not* a variable
  - Of course, the contents of the four char spaces may vary
    ```c
    m[2] = 42;
    ```
typedef

- A convenient way of abbreviating type names
- Usage: keyword `typedef`, followed by type definition, followed by new type name

```c
typedef char *ptr_to_char;
ptr_to_char p;  /* p is of type (char *) */
```
The keyword `const` makes the declared entity a constant rather than a variable:
It is given an initial value and then cannot be changed.

```c
int const a = 17;
```
The pointer `pa` will always point to the same address, but the data content at that address can be changed:

```c
*pa = 42;
```
Constant declarations

```c
int a = 17;
int b = 42;

int const * pa = &a;
```

- The pointer `pa` can be changed, but the data content that it’s pointing to cannot be changed:
  ```c
  pa = &b;
  ```
Constant declarations

```c
int a = 17;

int const * const pa = &a;
```

- Neither the pointer `pa` nor the data that it's pointing to can be changed.
If a variable is declared multiple times in a program, how many distinct variables are created?

Local variable declared within a function: a fresh instance of the variable is created – even if there’s a local variable in another function with exactly the same name.

There is no linkage here.
Linkage

- If a variable is declared multiple times in a program, how many distinct variables are created?
- Variables declared outside of any function: Only one instance of the variable is created (even if it’s declared in multiple files).
- This is **external** linkage.

```c
int a;  // external linkage
int a;
int f() { ... }
...

int a;  // external linkage
int g() { ... }
...
```
Forcing external linkage

- A local variable declared as `extern` has external linkage.

```c
int a;

int f ( void ) {
    extern int a;
}

int g ( void ) {
    extern int a;
}
```

Refer to the same variable

Declaring `a` here is not strictly necessary, since `f()` is within the scope of the first `a` declaration.
Dangers of external linkage

- It’s a way to avoid the trouble (both for the programmer and the machine) of passing parameters.
- But... it can lead to trouble, especially in large multi-file programs constructed by many people
  - Where exactly is the variable `a` declared?
  - What is all that other code (possibly in different files) doing with `a`?
  - If I modify `a` in a certain way, is it going to mess up code elsewhere that uses `a`?
  - It’s harder to reuse `g()` if it depends on a variable declared elsewhere
Restricting external linkage

- Q: What if you have a “global” variable, but you only want internal linkage (i.e. just within the file)?
- A: Declare it static:

```
static int a;
int f ( void ) {
    extern int a;
}
```

```
static int a;
int g ( void ) {
    extern int a;
}
```
Storage class: automatic

- If a variable declaration is executed multiple times, is new memory for the variable allocated each time?
- For automatic variables (what we’re accustomed to), the answer is “yes”.

```c
int f ( void ) { int temporary; ... }
```

- Each time `f()` is called, new memory is allocated for `temporary`. And every time a call to `f()` terminates, the memory is deallocated – that instance of `temporary “vanishes”`.
- All that “housekeeping” takes time and effort
Storage class: static

- If a variable declaration is executed multiple times, is new memory for the variable allocated each time?
- For static variables the answer is “no”. Memory is allocated once – at the first use of the variable – and then reused.

```c
int f ( void ) { static int persistent; ... }
```

- The first time `f()` is called, new memory is allocated for `persistent`.
- And every subsequent call to `f()` reuses that memory – potentially using values that earlier calls to `f()` left behind.
Why use static storage?

- Avoid overhead of allocating, initializing, deallocating memory with each function call
- Maintain some state information over multiple calls to the function

```c
int f( void ) {
  /* count number of times f has been called */
  static int num_calls = 0;
  ...
  num_calls++;
  return;
}
```
Confused about static?

- Yes, that’s right – `static` means two different things:
  - For “global” variables, declared outside of any function, static means “restrict the linkage of this variable to internal linkage”.
  - For “local” variables, declared inside a function, static means “allocate static memory for this variable”.

```