CMSC 216
Introduction to Computer Systems
Lecture 3
Introduction to C, cont.

Jan Plane & Alan Sussman
{jplane, als}@cs.umd.edu

Administrivia

- Project 1 posted tomorrow, due Sept. 22
  - public tests available in public Grace directory soon
- Read Chapters 4 and 5 of Reek

Chapter 3, Reek

DATA

Data types in C

- C has four basic types: integers, floating point numbers, pointers, and aggregate types (e.g., arrays and structures)
  - The first three are considered scalar types
- Integers include characters as well as the numbers you commonly think of
- Integer types have signed and unsigned versions
- Floating point types have (mostly) the same names as in Java: **`float`** and **`double`**

Data sizes in C

Type name	Minimum size	Size on `linux.grace`
`char`	1	1
`short`	2	2
`int`	2	4
`long`	4	8
`float`	4	4
`double`	8	8
`long double`	10	12

All sizes are in bytes.

Note that variables of type **`char`** are guaranteed to always be one byte.

There is no maximum size for a type, but the following relationships must hold:

```
sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
```


Numeric literals

- Suffixes allow you to specify a number is of a given type:
  - **`30000`** is of type **`int`**, but **`30000L`** is of type **`long`**
  - **`2.5`** is of type **`double`**, but **`2.5f`** is of type **`float`**
- Can also give numbers in different bases:
  - Precede with a zero for octal: **`017 == 15`**
  - Precede with **`0x`** for hexadecimal: **`0x1f == 31`**

Enumerated types

- Can use these to represent things that only take on certain values
- Values equal to 0, 1, 2... based on order of definition
- Values can be set by programmer to things other than 0, 1, 2...
- Based on integers, but don't mix enums with integers unless absolutely necessary

```
#include <stdio.h>
int main() {
  enum Suit {
    SPADES, HEARTS,
    DIAMONDS = 42, CLUBS
  };
  enum Suit suit1, suit2;

  suit1 = SPADES;
  suit2 = CLUBS;

  if (suit1 < suit2)
    printf("Spades are first.\n");
  else
    printf("Clubs are first.\n");
  printf("Spades = %d, Clubs = %d\n",
         suit1, suit2);
  return 0;
}
```


Variable declaration, initialization

- Declaring a variable provides space for it in memory
  - "**`int x;`**" will cause 4 bytes on the stack to be reserved for x
- No initialization takes place! And unlike Java nothing will prevent you from using that random value.
- Consider this code:

```
int x;
printf("X: %d\n", x);
```

- It might print **`X: 0`**
- It might also print **`X: -2349235`**, or **`X: 82373`**
- It's up to you to initialize your variables properly
- Initialization can be done at the same time as variable declaration:

```
int x = 42;
```

Identifier scope

- C has two main types of scope
  - Block scope: a variable declared inside a block is visible only within the block (includes nested blocks inside that block)
  - File scope: an identifier declared outside of any block is visible everywhere in the file **after** the declaration
    - applies both to global variables and function names

Identifier linkage

- What happens if we encounter two instances of the same identifier across different files?
- A function named **`foo()`** in **`file1.c`** can cause problems if there's a function named **`foo()`** in **`file2.c`**
- We can resolve these types of conflicts by changing the linkage of the functions
- Linkage: a property of an identifier that determines if multiple declarations of that identifier refer to the same object

Identifier linkage, cont.

- Three types of linkage
  - none: all declarations of an identifier refer to different entities (i.e., one copy per declaration)
  - internal: all declarations of an identifier inside a given file refer to the same entity, but declarations across files refer to different entities (i.e., one copy per file)
  - external: all declarations of an identifier refer to the same entity (i.e., one copy per program)
- Default linkage is different for different types of identifiers
  - all functions, and variables with file scope default to external linkage
  - variables with block scope default to no linkage
- Use **`extern`** and **`static`** to modify linkage

Linkage example

file1.c:

```
int bar(int);
extern int even_flag;

static int foo(int i) {
  return 2 * i;
}

int main() {
  even_flag = 0;
  printf("foo(2): %d\n", foo(2));
  printf("bar(2): %d\n", bar(2));
  if (even_flag)
    printf("even int passed to bar()\n");
  return 0;
}
```

file2.c:

```
int even_flag;

static int foo(int i) {
  return 3 * i;
}

int bar(int i) {
  if (i % 2 == 0)
    even_flag = 1;
  return foo(i);
}
```

Storage

- How long is memory allocated for an object?
- After this function returns, is there any guarantee that the value 5 will stay at the spot in memory at which it was stored?

```
int example(int i) {
  int j = 5;
  return i + j;
}
```

- There are two types of storage: static and automatic; the variable **`j`** above has automatic storage, meaning it is no longer maintained after its function returns

Storage, cont.

- Static storage means that the variable exists throughout the entire life of the program
  - global variables have this kind of storage
- Automatic storage is the default for block-scoped variables, but this can be changed with the **`static`** keyword
- Initializations to static variables occur only once

Storage examples

Program:

```
#include <stdio.h>

void foo() {
  static int i = 1;
  int j = 1;

  i = i + 1;
  j = j + 1;
  printf("i: %d; j: %d\n", i, j);
}

int main() {
  foo();
  foo();
  foo();
  return 0;
}
```

Output:

```
i: 2; j: 2
i: 3; j: 2
i: 4; j: 2
```


Chapter 4, Reek

STATEMENTS

Assignment statements

- Any expression can appear as a statement
- An assignment is just an expression (typically used as an expression statement)
  - The **=** is an *operator* in C (and right associative)
  - legal expression statements, assuming **`int x, y;`**

```
x = 3;
y + 3;
x == y;
```

- An expression statement is useful when the expression has a *side effect*
- An assignment returns a value - whatever value was assigned to the variable on the left hand side of the assignment

```
y = x = 123;
y = 5 + (x = 3);
```


C control statements

- These are very similar to Java, with one important difference: C has no boolean type
  - scalar expressions used instead; 0 is false, everything else is true
- C has **`if`**/**`else`**, **`while`**, **`for`**, **`do`**-**`while`**, and **`switch`** statements, just as Java does
  - but can't declare variables in **`for`** loop header
- **`break`**: immediately end loop
- **`continue`**: skip remainder of loop body, return to beginning of loop
  - in case of **`for`** loop, perform increment
- don't abuse **`break`**/**`continue`**, and rarely use **`continue`** if at all

Perfectly valid boolean examples

```
int c = ???;
if (c)
  f1();
if (c != 0)
  f2();
if (c == 2)
  f3();
if (c = 2)
  f4();
```

Function	-1	0	1	2
`f1()`	Y	N	Y	Y
`f2()`	Y	N	Y	Y
`f3()`	N	N	N	Y
`f4()`	Y	Y	Y	Y

The table shows which functions will be executed if certain values are assigned to **`c`** during its initialization. **`f4()`** always executes! Why?

Chapter 5, Reek

OPERATORS

Basic arithmetic operators

- Most of the operators you know from Java are in C
  - **+** adds, **-** subtracts, **/** divides, **%** performs remainder (modulus), ***** multiplies
- C also performs integer division (rather than floating point division) when both operands are integers
  - So, **`3.0 / 2.0 == 1.5`** (also **`== 3 / 2.0`**)
  - But, **`3 / 2 == 1`**

Numeric base conversion

- Computer only works in binary (base 2)
- People generally prefer decimal (base 10), but sometimes hexadecimal (base 16) is also useful
- Converting between these representations is important for us - hex is easily translated to binary, but decimal to/from either hex or binary can be a bit challenging
- It pays to memorize a few powers of 2

Simple conversion table

Decimal	Hex	Binary	Decimal	Hex	Binary
0	**0x0**	**0000**	8	**0x8**	**1000**
1	**0x1**	**0001**	9	**0x9**	**1001**
2	**0x2**	**0010**	10	**0xA**	**1010**
3	**0x3**	**0011**	11	**0xB**	**1011**
4	**0x4**	**0100**	12	**0xC**	**1100**
5	**0x5**	**0101**	13	**0xD**	**1101**
6	**0x6**	**0110**	14	**0xE**	**1110**
7	**0x7**	**0111**	15	**0xF**	**1111**