CMSC 216
Introduction to Computer Systems

Lecture 4
Introduction to C and arrays

Administrivia

- Read Chapter 5 of Reek
  - parts of Chapters 8 and 9 covered today and Tuesday

Chapter 5, Reek

Operators (Continued)

Compound assignment

- C supports several compound assignment operators that can save you time and typing
  - include **+=**, **-=**, ***=**, **/=**, **%=**, **<<=**, **>>=**, **&=**, **|=**, **^=**
  - does NOT include **!=**
- Can reduce possibility of errors:

```
a[i * j + k / 2] = a[i * i + k / 2] + 10;
a[i * j + k / 2] += 10;
```

- But be careful:

```
a[f(b) % n] = a[f(b) % n] + 1;
```

vs.

```
a[f(b) % n] += 1;
```

Increment/decrement operators

- These work just like they do in Java
- Remember the difference between **`++i`** and **`i++`**?
- What does this function output?

```
void foo() {
   int i = 10, j = 5;
   printf("%d\n", --i);
   printf("%d\n", j++);
   printf("%d\n", i++ + j);
}
```

Other unary operators

- **(*typename*)** is a unary operator
  - Works just as in Java
- There is also a **–** operator which performs arithmetic negation
  - so code like "**`a *= -1;`**" is really just wasteful
- We'll discuss the unary **`&`** and **`*`** operators soon, when we discuss pointers

The `sizeof` operator

- Unary operator, evaluates to the number of bytes necessary to hold its operand
- Operand can be an expression or a type name
- Does NOT evaluate the expression
- Examples:

```
int i = 5;
printf("%d\n", sizeof(i));
printf("%d\n", sizeof(unsigned char));
printf("%d\n", sizeof(i++));
printf("%d\n", i);
```

Boolean operators

- Relational operators: `<`, `>`, `<=`, `>=`
- Equality operators: `==`, `!=`
- Logical operators: `&&`, `||`, `!`
- Function just as you'd expect from working in Java, except that they evaluate to 1 (if true) or 0 (if false), so this example actually makes sense:

```
int i;
i = (! 3) == (4 < 2);
i = (! 2) || (5 && i);
```

- Remember that the logical operators do short-circuit, affecting whether or not parts of expressions get evaluated

Conditional operator

- The only ternary operator
- Syntax: *expr1* **?** *expr2* **:** *expr3*
- If expr1 is nonzero, evaluates to expr2; otherwise, evaluates to expr3
- Don't abuse this; use it only when it helps reduce code duplication:

```
if (a > 5)
  b[2 * c + f(d / 5)] = 3;
else
  b[2 * c + f(d / 5)] = -20;
```

Comma operator

- Yes, the comma is an operator
- Evaluates left operand, then right operand
- Value of expression with comma is value of last operand
- Has lowest precedence of all operators
- So what gets stored in **`i`** after each statement? What does each statement evaluate to?

```
i = 1, 2, 3, 4;
i = (1, 2, 3, 4);
```

Precedence and associativity

- Different operators can fall on different precedence levels
- Ties among levels are settled by associativity rule for that level
- Some operators impose restrictions on evaluation order, but aside from that, compiler can optimize
- Full table in *Pointers on C*, pgs. 114-115

Lvalues and Rvalues

- An rvalue is anything that can appear on the right side of an assignment statement
  - virtually any expression
- An lvalue is anything that can appear on the left side of an assignment statement
  - values that represent a place to store a value
- The right and left sides of an assignment statement are treated differently
  - right hand side is a value, left hand side is a location to store a value (an address)

Implicit type conversion

- Arithmetic operators require their operands to be of the same type to perform the operation
- **`int`** is actually “smallest” type used
- There is a hierarchy of types (Reek, § 5.4.2)
  - floating-point numbers over integers
  - wide over small
  - unsigned over signed
- Operation result is of the new type
- What to do? **`2000000000 * 3`**

Mixed-type assignments

- RHS is converted to LHS type before storage
- Can mean either promotion or truncation

```
char a, b = 'b', c = 'c';
float f = 2.25, g = 4.9999;
unsigned int i, j;
unsigned char ch;
a = b + c; /* a = 98 + 99 = 197? */
i = f; j = g; /* i: 2; j: 4 */
i = ch = 0xabcd; /* i: 0xcd; ch: 0xcd */
```

Parts of Chapters 8 and 9, Reek

Arrays and Strings

Arrays in C (Static Type)

- Much like Java (or many other languages)
  - All elements are the same type
  - Elements indexed by the subscript operator `[]`
- Sizes must be known at compile time (constant expressions only) and are static
- Can't assign to arrays (can initialize, though)
- Can use `==` and `!=`, but meaning isn't what you might think (wait until pointers)
- Syntax for creating arrays is slightly different
  - C: `int a[5];` creates array of 5 `int`s
  - Java: `int[] a = new int[5];` creates array of 5 `int`s

Array initialization

- Supply a list of values in braces, separated by commas:

```
int a[5] = {1, 1, 2, 3, 5};
```

- Occurs when array is first created
  - when this occurs depends on the array's storage class
  - also means you can't initialize after declaration
  - and you can't initialize with variable expressions
- Zeroes pad the array when initializer is short
- Use of an initializer allows size to be omitted
  - can't omit size otherwise when declaring local variables (parameters are an exception, though)
- Initializers with excess elements cause errors

Array initialization examples

```
int a[3]    = {1, 4, 7};
int b[5]    = {2, 8};
int c[]     = {3, 9, 5, 2 + 6};
int d[1000] = {0};
-----------------------------------
```

(assuming `i` is an `int` variable):

```
int w[i];  /* both illegal: size must be */
int x[];   /*      known at compile time */

int y[4] = {2 * i, 3 + 2};  /* illegal: initializer
                values must be known at compile time */

int z[5];  /* this OK */

z = {1, 1, 2, 3, 5}; /*illegal: initializer can only
            be used as it first comes into existence */
```

Parameters in C

- In C, all variables are passed by value – a copy of the argument is put into the parameter
  - Modifying the parameter does not change the argument
- The name of an arrays is actually a reference to the 0^{th} element of the array.

Array parameters

- So array parameters act as if they were passed by reference
- If a function modifies elements of an array parameter, the array passed in **is** modified

```
void function(int a[]);
...
int array[10];
function(array);
```

- Sizes for array parameters are ignored – only types matter
  - so "`void function(int a[12397]);`" is equivalent to the above prototype
- Generally: You must pass the size as an additional argument

Use of symbolic constants

- #define preprocessor directive

  `#define name value`
- All occurrences of **`name`** in the source file are replaced by **`value`**
- Used to define constants for things such as array sizes and other values to improve program maintainability

```
#define ARR_SIZE 3
int main() {
  int i, a[ARR_SIZE] = {1, 2, 3};
  multiply_array(5, a, ARR_SIZE);
  printf("[%d", a[0]);
  for (i = 1; i < ARR_SIZE; i++)
    printf(", %d", a[i]);
  printf("]\n");
  return 0;
}
```

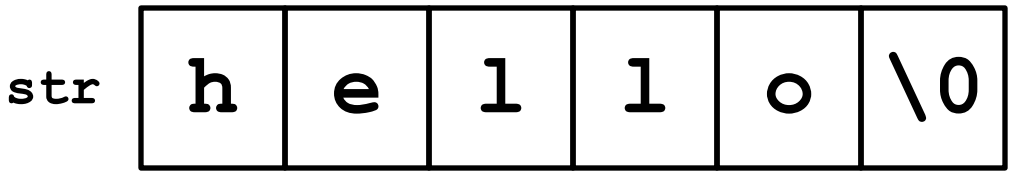
Strings in C

- There is no String type in C
- In C, a string is defined as a sequence of characters that is followed by a byte with the value zero
  - often called: "zero byte", "null byte", "NUL"
  - represented as the character literal `'\0'`
  - "null byte" is NOT the same thing as "null pointer"
- Since arrays are contiguous in memory, and **`char`**s are all one byte in size, we can use arrays of **`char`**s to hold strings
- **`printf()`** format specifier for strings is **`%s`**

String initialization

- Because character arrays are so closely related to strings, they can be initialized with string literals as well as standard array initializers
- But don't forget that the null byte needs to be stored as well
- Example:

```
char str[6] = "hello";
```



More examples of string initialization

```
char a[] = "hello";
char b[10] = "Maryland";
char c[1024] = "";
```