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? \( 2000000000 \times 3 \)

Mixed-type assignments

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

```c
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 */
```

Arrays in C

• 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 ints
  – Java: `int[] a = new int[5];` creates array of 5 ints
Array initialization

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

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

These are illegal (assuming i is an int):
```c
int w[i];
int x[];
int y[4] = {2 * i, 3 + 2};
int z[5]; /* this alone is OK */
z = {1, 1, 2, 3, 5};
```

Parameters in C

- You can use parameters as variables, but why is it safe?
- In C, variables are passed by value – a copy is passed

```c
int abs_value(int x) {
    if (x < 0)
        x = -x;
    return x;
}

int main() {
    int n = -17, a;
    a = abs_value(n);
    printf("%d %d\n", a, n);
    return 0;
}
```

Array parameters

- Array parameters act as if they were passed by reference (as we’ll discuss later)
- If a function modifies elements of an array parameter, the array passed in is modified
  ```c
  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
Array parameters, cont.

- You generally have to pass array size along with the array.
- Functions only know about the elements of an array – the size of an array parameter isn’t known.

```c
void multiply_array(int factor, int arr[], int ct) {
  int i;
  for (i = 0; i < ct; i++)
    arr[i] *= factor;
}

int main() {
  int a[] = {1, 2, 3};
  multiply_array(5, a, 3);
  printf(“[%d, %d, %d]
”, a[0], a[1], a[2]);
  return 0;
}
```

Use of symbolic constants

- `#define` preprocessor directive
  ```c
  #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.

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

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

  str: h e l l o \0
  ```
More examples of string initialization

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

Basic string library functions

- C has many different functions for working with strings; to use these, you must
  ```c
  #include <string.h>
  ```
- We're only covering a small subset here; if you ever want to see all of them, more
  information can be found in the `string.h` man page
  - Note: the prototypes there are slightly different than what we'll be covering here, because we
    haven't covered pointers yet, but functionality is the same

String library functions, cont.

- String length:
  ```c
  size_t strlen(char str[]);
  ```
  - returns the length of the string pointed to by the
    string passed in as a parameter
  - string length is the number of characters in the
    string, not counting the null byte
  - Example:
    ```c
    char str[] = "ice cream";
    printf(""\"\s\": %d chars\n", str, strlen(str));
    Output:
    "ice cream": 9 chars
    ```

A possible `strlen()` implementation

```c
size_t strlen(char str[]) {
    size_t i;
    for (i = 0; str[i]; i++)
    {
        return i;
    }
```

- The integer type `size_t` is defined in the
  standard library header file, and is big enough to
  hold the size of any type
- What would happen if you passed an
  uninitialized character array into this function?
String library functions, cont.

• Comparing strings:

```c
int strcmp(char s1[], char s2[]);
```
– works just the same as `s1.compareTo(s2)` did in Java:
  • returns negative number if `s1` is less than `s2`
  • returns positive number if `s1` is greater than `s2`
  • returns 0 if `s1` and `s2` match character for character

– Example:

```c
if (strcmp(str1, "hello") == 0)
    printf("str1 is \"hello\"\n");
```