Functions

Class 15

Functions

- new versions of C (and C++) include the function strtod
- a robust, error-checking function that converts a string to the equivalent double value
- to illustrate functions in C89, let us write atod (ASCII to double)
- compare atof in K&R page 71
- look at code atod.c

Differences with K&R

- the incoming string parameter is named "string" instead of "s"
- its type is const char* rather than char[] to emphasize that this function will not change the actual parameter
- longer, more descriptive variable names for val, power, and i
- one variable per declaration
- any variables that need to be initialized before use are initialized at declaration
  (Note! do not unnecessarily initialize a variable)
- all three loops are while loops that continue until a condition is met after an unpredictable number of iterations
- if the number of iterations can be determined at the start of the loop, use a for loop, otherwise, use while or do-while
- all control constructs use braces
- notice that every binary operator is space-separated

Declaration vs. Definition

- a declaration declares the name and type of a thing (variable, or function)
- a definition causes storage to be allocated for the thing

- a thing can be declared many times
- a thing must be defined exactly once

- most of the time (so far) a variable is both declared and defined in one statement
- in C++, a function must be declared (prototype) separately from its definition (body)
- in C, a function should be declared separately from its definition (and often it must be anyway)

Local and Global

- a variable (or formal parameter) declared within a function is local to that function
- a variable declared outside the scope of a function is a global variable
- global variables today are understood to be unsafe and to represent bad programming practice
- they still exist and we must understand them

Local and Global

```
int foo; /* this is a global variable */

void bar(void)
{
  int bam; /* a local variable */

  foo ... /* foo is visible here */
  bam ... /* bam is visible here */

  /* qux is NOT visible here */
}

int main(void)
{
  int qux; /* a local variable */
  foo ... /* foo is visible here */

  /* bam is NOT visible here */
}
```

Storage Classes

- C has three main storage classes for variables
  - auto
  - extern
  - static
- (there's also a fourth, register, but clang is so much smarter than you are about C that even if you use it, clang will probably ignore it anyway)

- auto is if you don't specify either extern or static
- C++ uses auto differently
- we will never directly use auto

Linkage Categories

- most real programs comprise many source code files
- each .c file is compiled separately into a .o file
- the linker is responsible for stitching all the .o files (and library objects) together into a single executable

- C supports three linkage categories
  - external: available to all files in a program: all functions are automatically external, and normal global variables are
  - internal: available only within the file in which it is declared and defined: static global variables
  - none: available only within its own block: local variables

extern

- extern is used to declare that a variable is defined with external linkage somewhere in the project
- a variable can only be defined once, in one file
- if the variable is global, it must be declared using extern in any file that needs access to it

- the exact same rule applies to functions
- except that since extern is automatic for functions, using the word extern is redundant

extern Example

File 1

```
extern int x;
extern int y;
int x = 10;

void foo(void);
void bar(void);

int main(void)
{
  foo();
  bar();
  return 0;
}

void foo(void)
{
  x += y;
}
```

File 2

```
extern int x;
extern int y;
int y = 23;

void bar(void);

void bar(void)
{
  x += y;
}
```

static Variables

- static variables have permanent storage allocated to them (as long as the program is running)
- static works differently for local and global variables

static Local Variables

- normal local variables come into existence when a function is called
- they do not exist before or after the function execution
- if a function is called a second time, the previous value a local variable had is lost

- a static local variable has a permanent storage location
- that location is unavailable when the function is not executing
- but the value in it is preserved between two calls of the function

static Local Variable Example

```
1  unsigned get_next_in_series(void)
2  {
3    static unsigned value = 0;
4
5    value += 11;
6    return value;
7  }
```

- the initialization on line 3 happens only once, the first time the function is called
- all other times the function executes, value retains its value from the previous execution
- this allows you to have a globally available counter without a global variable

static Global Variables

- a static global variable's access is restricted to the file in which it is declared and defined
- it's a "local" global variable
- perhaps a bit safer than a "global" global variable
- but still not good for general use

- note: clang requires you to specify extern or static on every global variable

- a valid use: need the previous counter, but need to be able to initialize it at runtime rather than compile time

static Global Variable Example

```
static unsigned value;
static unsigned step_size;

unsigned get_next_in_series(void);
void initialize_series(unsigned seed, unsigned step);

unsigned get_next_in_series(void)
{
  value += step_size;
  return value;
}

void initialize_series(unsigned seed, unsigned step)
{
  value = seed;
  step_size = step;
}
```