CS61B Lecture #27

Administrative:

Today:

- Background on the C language
- C Program Structure

The C Programming Language: Background

- Predecessor of Java, C++
- Comes from chain of systems-programming languages:
  - BCPL (ca. 1967) $\Longrightarrow$ B (ca. 1970) $\Longrightarrow$ NB (ca. 1971) $\Longrightarrow$ C (ca. 1972-3).
- Need came from implementation of Unix (on small machines, starting with a PDP-7 with about 40KB (!) of memory).
- Language continued to evolve. Standardization effort started in 1983 (ANSI X3J11 Committee).
- ANSI standard report in 1989, later becomes ISO standard.
- Programs exist that use earlier versions of the language, which has remained largely backwards compatible.
- I will pretend that only the modern syntax exists for the time being.

Major Omissions

- Java adopted most of *C*'s syntax for expressions, control statements (**while**, **if**, etc.), but *C* is much more spare.
- No object-oriented features (instance methods, dynamic type queries)
- No access control (public/private) to speak of.
- No reflection (nothing like `Class` type), no dynamic loading of classes
- No overloading.
- Threads not built in (use various libraries instead).
- Exceptions not built in (special library functions instead).

Additions and Other Differences

- Program structure organized around functions and global variables rather than classes.
- Memory model different:
  - *can* have pointers to named objects,
  - *can* have names on things other than primitive types and pointer variables.
  - *can* pass values other than primitives and pointers.
- Heap allocation not built in; all memory management explicit.
- Union types, enumerated types.
- Many more casts are legal; can create pointers from scratch.
- No boolean type: use `int`.

C Program Structure

- Classes (called "structs") don't contain functions, just fields.
- All functions are "bare," corresponding to Java's static functions, but without class name.
- Program is just a collection of declarations or definitions of functions, types, and what Java would call static variables.
- One function called '`main`' is the starting point for execution.

Functions

- Can both *declare* and *define* functions.
  - Declaration looks like a Java abstract method declaration:

    ```
    extern int global (int x, double y);
    extern int globalVar;
    static void local (void);  /* No arguments */
    static int staticVar;
    ```

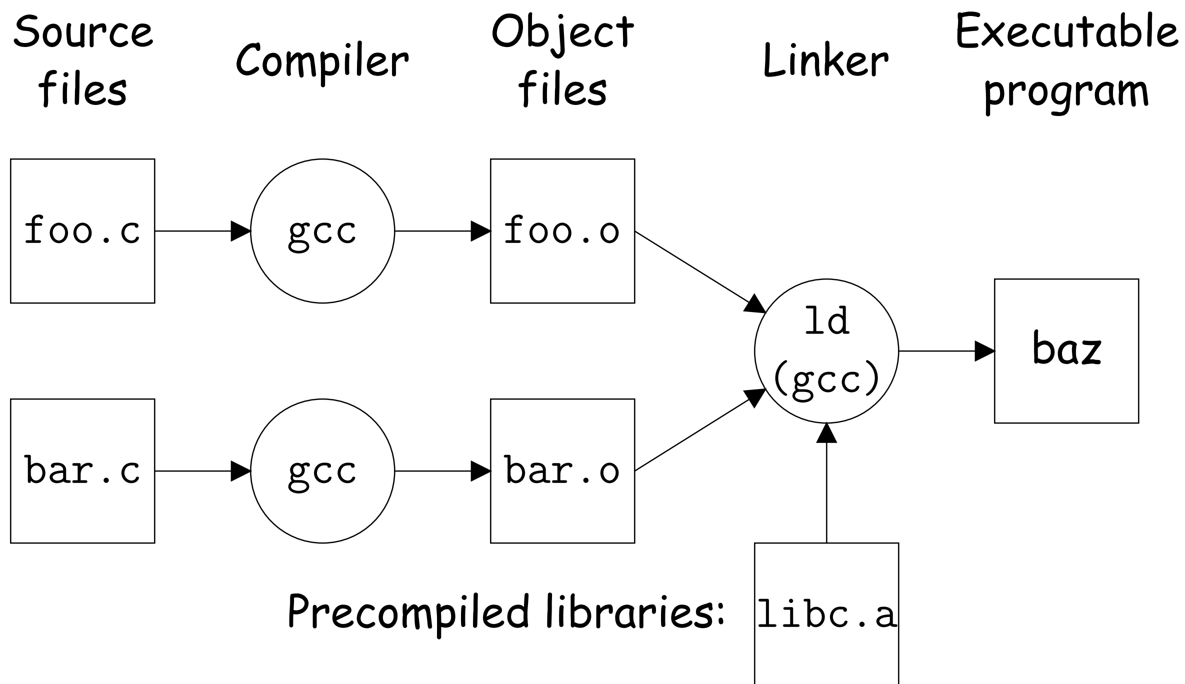
    means these functions are defined somewhere. Can repeat this declaration as often as you like.
  - Definition is like ordinary Java method declaration:

    ```
    int global (int x, double y) { ... }
    int globalVar = 42;
    static void local (void) { ... }
    static int staticVar = 10;
    ```

  - `global` has *external linkage,* available everywhere in program
  - `local` has *internal linkage,* available only in the the same compilation (roughly, the same file, like Java private declarations)

Separate Compilation

- Typically, systems compile one source file (`foo.c`) at a time, and then *link* results together:



- Each source file compiled independently.
- In contrast to `javac`, typical C compiler does not go looking for functions in other source files.
- So, there is issue with consistency; everyone needs same idea of what the parameters to `f` are.

Separate Compilation

- Declarations are the answer:

```
foo.c:                          |  bar.c:
                                |
extern void barFunc (int x);    |  extern void fooFunc (double
                                |
int fooFunc (double y) {        |  void barFunc (int x) {
  ...                           |    ...
  barFunc (3);                  |    fooFunc (3.14)
}                               |  }
```

- But there's still no guarantee that both `foo` and `bar` will agree on the declarations.
- Duplicating declarations for `barFunc` and `fooFunc` in all files that use them is error-prone.

Header Files

- Solution: centralize declarations in header files, used by all programs:

```
foo.c:                          |  bar.c:
                                |
#include "baz.h"                |  #include "baz.h"
                                |
int fooFunc (double y) {        |  void barFunc (int x) {
  ...                           |    ...
  barFunc (3);                  |    fooFunc (3.14)
}                               |  }
------------------------------------------------------
             baz.h:

             extern void barFunc (int x);
             extern void fooFunc (double y);
```

- `#include "baz.h"` simply inserts contents of `baz.h`.
- Same mechanism used for declarations of things in precompiled libraries:

```
#include <stdio.h>

main () {
 printf ("Hello, world!\n");  /* From stdio.h */
}
```

Example: Primes

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int isPrime (int x) {
  if (x == 2)
    return 1; /* True */
  else if (x < 2 || x % 2)
    return 0; /* False */
  else {
    int k, lim = 1 + (int) sqrt (x);
    for (k = 3; k < lim; k += 2)
      if (x % k == 0)
        return 0;
    return 1;
  }
}

main (int argc, char* argv[]) {
  int N = argc == 2 ? atoi (argv[1]) : 0;
  if (isPrime (N))
    printf ("%d is prime.\n", N);
  else
    printf ("%d is not prime.\n", N);
  exit (0);
}
```