CSE 303: Concepts and Tools for Software Development

Hal Perkins

Autumn 2007

Lecture 13— C: post-overview, function pointers; Makefiles

Where are We

Today:

- Top-down view of C
- Function pointers
- The `make` program (not C-specific)

Later:

- Using function pointers more like objects

Top-down post-overview

Now that we have seen most of C, let's summarize/organize:

- Preprocessing (text replacement; common conventions)
  - `#include` for declarations defined elsewhere
  - `#ifdef` for conditional compilation
  - `#define` for *token-based* textual substitution
- Compiling (type-checking and code-generating)
  - A sequence of *declarations*
  - Each C file becomes a `.o` file
- Linking (more later)
  - Take `.o` and `.a` files and make a program
  - `libc.a` in by default, has printf, malloc, ...
- Executing (next slide)

Execution

- O/S maintains the “big array” address-space illusion
- Execution starts at `main`
- Each stack-frame has space for arguments, locals, and return-address (last one shouldn’t be visible to you)
- Library manages the heap via `malloc`/`free`

C, the language

- A file is a sequence of *declarations*:
  - Global variables (`t x;` or `t x = e;`)
  - `struct` (and `union` and `enum`) definitions
  - Function *prototypes* (`t f(t1,...,tn);`)
  - Function definitions
  - `typedefs`
- A function body is a *statement*
  - Statements are much like in Java (+ `goto`, – exception-handling, ints for bools, ...)
  - Local declarations have local scope (stack space).
- Left-expressions (locations) and right-expressions (values, including pointers-to-locations)
  - * for pointer dereference, & for address-of, . for field access

C language continued

“Convenient” expression forms:

- `e->f` means `(*e).f`
- `e1[e2]` means `*(e1 + e2)`
  - But + for pointer arithmetic takes the size of the pointed to element into account!
  - That is, if `e1` has type `t*` and `e2` has type `int`, then , then `(e1 + c) == (((int)e1) + (sizeof(t) * c))`
  - The compiler “does the sizeof for you” – don’t double-do it!

“Size is exposed”: In Java, “(just about) everything is 32 bits”. In C, pointers are usually the same size as other pointers, but not everything is a pointer.

New side point: padding, alignment may mean structs are “bigger than expected”

C is unsafe

The following is allowed to do *anything* to your program (delete files, launch viruses, silently turn a 3 into a 2, ...)

array-bounds violation (bad pointer arithmetic), dangling-pointer dereferences (including double-frees), dereferencing NULL, using results of wrong casts, using contents of uninitialized locations, linking errors (inconsistent assumptions), ...

Pointer casts are not checked (no secret fields at run-time; all bits look the same)

Often crashing is a “good thing” compared to continuing silently with meaningless data.

Now

C is a pretty small language, but we still skipped lots of features.

For now, one *idiom* (returning error codes) and one useful *feature* (function pointers).

Error codes

Without exceptions, how can a callee indicate it could not do its job?

- Through the return value; caller *must remember to check*

Examples:

- `fopen` may return NULL
  - `f=fopen("someFile","r"); if(!f) ...`
- `scanf` returns number of matched arguments
  - `cnt=scanf("%d:%d:%d",&h,&m,&s); if(cnt!=3) ...`
- Often assign "real results" through pointer-arguments and result is 0 for success and other values for errors (like in bash)
  - `if(!someCall(&realAns,arg1,args)) ...`

Function pointers

“Pointers to code” are almost as useful as “pointers to data”.

(But the syntax is more painful.)

(Somewhat silly) example:

```
void app_arr(int len, int * arr, int (*f)(int)) {
  for(; len > 0; --len)
    arr[len-1] = (*f)(arr[len-1]);
}
int twoX(int i) { return 2*i; }
int sq(int i) { return i*i; }
void twoXarr(int len, int* arr) { app_arr(len,arr,&twoX); }
void sq_arr(int len, int* arr) { app_arr(len,arr,&sq); }
```

CSE 341 spends a week on *why* function pointers are so useful; today is mostly just *how* in C.

Function pointers, cont'd

Key computer-science idea: You can pass what code to execute as an argument, just like you pass what data to process as an argument.

Java: An object is (a pointer to) code *and* data, so you're doing both all the time.

```
// Java
interface I { int m(int i); }
void f(int arr[], I obj) {
   for(int len=arr.length; len > 0; --len)
     arr[len-1] = obj.m(arr[len-1]);
}
```

The `m` method of an `I` can have access to data (in fields).

C separates the *concepts* of code, data, and pointers.

C function-pointer syntax

C syntax: painful and confusing. Rough idea: The compiler "knows" what is code and what is a pointer to code, so you can write less than we did on the last slide:

```
arr[len-1] = (*f)(arr[len-1]);
  → arr[len-1] = f(arr[len-1]);
app_arr(len,arr,&twoX);
  → app_arr(len,arr,twoX);
```

For types, let's pretend you always have to write the "pointer to code" part (i.e., `t0 (*)(t1,t2,...,tn)`) and for declarations the variable or field name goes after the `*`.

Sigh.

Onto tools

The language-implementation (preprocessor, compiler, linker, standard-library) is hardly the only useful thing for developing software.

The rest of the course:

- Tools (recompilation managers, version control, debuggers, profilers)
- Software-engineering issues
- A taste of C++
- Concurrency
- Societal implications

make

`make` is a classic program for controlling what gets (re)compiled and how. Many other such programs exist (e.g., `ant`, “projects” in IDEs, ...)

`make` has tons of fancy features, but only two basic ideas:

1. Scripts for executing commands
2. *Dependencies* for avoiding unnecessary work

To avoid “just teaching make features” (boring and narrow), let’s focus more on the concepts...

Build scripting

Programmers spend a lot of time "building" (creating programs from source code)

- Programs they write
- Programs other people write

Programmers automate repetitive tasks. Trivial example:

```
gcc -Wall -g -o myprog foo.c bar.c baz.c
```

If you:

- Retype this every time: "shame, shame"
- Use up-arrow or history: "shame" (retype after logout)
- Have an alias or bash script: "good-thinkin"
- Have a Makefile: you're ahead of us

“Real” build processes

On larger projects, you can’t or don’t want to have one big (set of) command(s) that redoes everything every time you change anything.

1. If `gcc` didn’t combine steps behind your back, you could need to preprocess and compile each file, then call the linker.
2. If another program (e.g., `sed`) created some C files, you would need an “earlier” step.
3. If you have other outputs for the same source files (e.g., `javadoc`), it’s unpleasant to type the source files multiple times.
4. If you want to distribute source code to be built by other users.
5. If you have $\mathbf{10^5}$ to $\mathbf{10^7}$ lines of source code, you don’t want to recompile them all every time you change something.

A simple script handles 1–4 (use a variable for the filenames for 3), but 5 is trickier.

Recompilation management

The "theory" behind avoiding unnecessary compilation is a "dependency dag":

- To create a target t, you need sources s_1,s_2,...,s_n and a command c (that directly or indirectly uses the sources)
- If t is *newer* than every source (file-modification times), assume there is no reason to rebuild it.
- Recursive building: If some source s_i is itself a target for some other sources, see if it needs to be rebuilt. Etc.
- Cycles "make no sense"

Theory applied to C

Another whole lecture on *linking* is in our future, but here is what you need to know today for C:

- Compiling a `.c` creates a `.o` and depends on all included files (recursively/transitively).
- Creating an executable ("linking") depends on `.o` files.
- So if one `.c` file changes, just need to recreate one `.o` file and relink.
- If a header-file changes, may need to rebuild more.
- Of course, this is only the simplest situation.

An algorithm

What would a program (e.g., a shell script) that did this for you look like? It would take:

- a bunch of triples: target, sources, command(s)
- a "current target to build"

It would compute what commands needed to be executed, in what order, and do it. (It would detect cycles and give an error.)

This is exactly what programs like `make`, `ant`, and things integrated into IDEs do!

make basics

The “triples” are typed into a “makefile” like this:

```
target: sources
        command
```

Example:

```
foo.o: foo.c foo.h bar.h
       gcc -Wall -o foo.o -c foo.c
```

Syntax gotchas:

- The colon after the target is required.
- Command lines must start with a **TAB NOT SPACES**
- You can actually have multiple commands (executed in order); if one command spans lines you must end the previous line with \.
- Which shell-language interprets the commands? (Typically bash, to be sure set the SHELL variable in your makefile.)

Using `make`

At the prompt:

```
prompt% make -f nameOfMakefile aTarget
```

Defaults:

- If no `-f` specified, use a file named `Makefile`.
- If not target specified, use the first one in the file.

Together: I can download a tarball, extract it, type `make` (four characters) and everything should work.

Actually, there's typically a "configure" step too, for finding things like "where is the compiler" that *generates* the `Makefile` (but we won't get into that).

Basics Summary

So far, enough for homework 4 and basic use.

- A tool that combines scripting with dependency analysis to avoid unnecessary recompilation.
- Not language or tool-specific: just based on file-modification times and shell-commands.

But there's so much more you want to do so that your Makefiles are:

- Short and modular
- Easy to reuse (with different flags, platforms, etc.)
- Useful for many tasks
- Automatically maintained with respect to dependencies.

Also, reading others' makefiles can be tough because of all the features: see `info make` or entire books.