Lecture 9

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Review: Void pointers

- Void pointer – points to any data type:

```
int x; void *px=&x; /* implicit cast to (void *) */
float f; void *pf =&f;
```

- Cannot be dereferenced directly; void pointers must be cast prior to dereferencing:

```
printf ( "%d %f\n", *( int *)px , *( float *)pf );
```

Review: Function pointers

- Functions not variables, but also reside in memory (i.e. have an address) – we can take a pointer to a function
- Function pointer declaration:

```
int (*cmp)(void *, void *);
```

- Can be treated like any other pointer
- No need to use `&` operator (but you can)
- Similarly, no need to use `*` operator (but you can)

Review: Function pointers

```
int   strcmp_wrapper (   void   * pa ,  void   * pb)   {
   return    strcmp ((  const   char   *)pa,    ( const   char   *)pb );
}
```

- Can assign to a function pointer:

  `int (*fp )(void *, void *) = strcmp_wrapper;` or

  `int (*fp )(void *, void *) = &strcmp_wrapper;`

- Can call from function pointer: (`str1` and `str2` are strings)

  `int ret =fp(str1, str2);` or

  `int ret =( *fp )( str1 , str2 );`

Review: Hash tables

- Hash table (or hash map): array of linked lists for storing and accessing data efficiently
- Each element associated with a key (can be an integer, string, or other type)
- Hash function computes hash value from key (and table size); hash value represents index into array
- Multiple elements can have same hash value – results in collision; elements are chained in linked list

6.087 Lecture 9 – January 22, 2010

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Symbols and libraries

- External libraries provide a wealth of functionality – example: C standard library
- Programs access libraries' functions and variables via identifiers known as *symbols*
- Header file declarations/prototypes mapped to symbols at compile time
- Symbols linked to definitions in external libraries during *linking*
- Our own program produces symbols, too

Functions and variables as symbols

- Consider the simple hello world program written below:

```
#include    <stdio .h>

const   char   msg[]   = "Hello, world.";

int    main ( void )   {
    puts (msg);
    return     0;
}
```

- What variables and functions are declared globally?

Functions and variables as symbols

- Consider the simple hello world program written below:

```
#include    <stdio .h>

const   char   msg[]   = "Hello, world.";

int    main ( void )   {
    puts (msg);
    return     0;
}
```

- What variables and functions are declared globally?
  `msg`, `main()`, `puts()`, others in `stdio.h`

Functions and variables as symbols

- Let's compile, but not link, the file hello.c to create hello.o:
  `prompt% gcc -Wall -c hello.c -o hello.o`
  - `-c`: compile, but do not link hello.c; result will compile the code into machine instructions but not make the program executable
  - addresses for lines of code and static and global variables not yet assigned
  - need to perform *link* step on `hello.o` (using `gcc` or `ld`) to assign memory to each symbol
  - linking resolves symbols defined elsewhere (like the C standard library) and makes the code executable

Functions and variables as symbols

- Let's look at the symbols in the compiled file hello.o:

```
prompt% nm hello.o
```

- Output:

```
0000000000000000 T main
0000000000000000 R msg
                 U puts
```

- 'T' – (text) code; 'R' – read-only memory; 'U' -undefined symbol
- Addresses all zero before linking; symbols not allocated memory yet
- Undefined symbols are defined externally, resolved during linking

Functions and variables as symbols

- Why aren't symbols listed for other declarations in `stdio.h`?
- Compiler doesn't bother creating symbols for unused function prototypes (saves space)
- What happens when we link?
  `prompt% gcc -Wall hello.o -o hello`
  - Memory allocated for defined symbols
  - Undefined symbols located in external libraries (like `libc` for C standard library)

Functions and variables as symbols

- Let's look at the symbols now:
  `prompt% nm hello`
- Output:
  (other default symbols)
  ⋮

```
0000000000400524 T main
000000000040062c R msg
                 U puts@@GLIBC_2.2.5
```

- Addresses for static (allocated at compile time) symbols
- Symbol `puts` located in shared library GLIBC_2.2.5 (GNU C standard library)
- Shared symbol `puts` not assigned memory until run time

Static and dynamic linkage

- Functions, global variables must be allocated memory before use
- Can allocate at compile time (static) or at run time (shared)
- Advantages/disadvantages to both
- Symbols in same file, other `.o` files, or static libraries (archives, `.a` files) – static linkage
- Symbols in shared libraries (`.so` files) – dynamic linkage
- `gcc` links against shared libraries by default, can force static linkage using `-static` flag

Static linkage

- What happens if we statically link against the library?

```
prompt% gcc -Wall -static hello.o -o hello
```

- Our executable now contains the symbol `puts`:

```
⋮
0000000000004014c0 W puts
⋮
0000000000400304 T main
⋮
0000000000046cd04 R msg
⋮
```

- 'W': linked to another defined symbol

1

Static linkage

- At link time, statically linked symbols added to executable
- Results in much larger executable file (static – 688K, dynamic – 10K)
- Resulting executable does not depend on locating external library files at run time
- To use newer version of library, have to recompile

Dynamic linkage

- Dynamic linkage occurs at run-time
- During compile, linker just looks for symbol in external shared libraries
- Shared library symbols loaded as part of program startup (before `main()`)
- Requires external library to define symbol exactly as expected from header file declaration
  - changing function in shared library can break your program
  - version information used to minimize this problem
  - reason why common libraries like `libc` rarely modify or remove functions, even broken ones like `gets()`

Linking external libraries

- Programs linked against C standard library by default
- To link against library `lib`*`namespec`*`.so` or `lib`*`namespec`*`.a`, use compiler flag `-l`*`namespec`* to link against library
- Library must be in library path (standard library directories + directories specified using `-L` *`directory`* compiler flag
- Use `-static` for force static linkage
- This is enough for static linkage; library code will be added to resulting executable

Loading shared libraries

- Shared library located during compile-time linkage, but needs to be located again during run-time loading
- Shared libraries located at run-time using linker library `ld.so`
- Whenever shared libraries on system change, need to run `ldconfig` to update links seen by `ld.so`
- During loading, symbols in dynamic library are allocated memory and loaded from shared library file

Symbol resolution issues

- Symbols can be defined in multiple places
- Suppose we define our own `puts()` function
- But, `puts()` defined in C standard library
- When we call `puts()`, which one gets used?

Symbol resolution issues

- Symbols can be defined in multiple places
- Suppose we define our own `puts()` function
- But, `puts()` defined in C standard library
- When we call `puts()`, which one gets used?
- Our `puts()` gets used since ours is static, and `puts()` in C standard library not resolved until run-time
- If statically linked against C standard library, linker finds two `puts()` definitions and aborts (multiple definitions not allowed)

Symbol resolution issues

- How about if we define `puts()` in a shared library and attempt to use it within our programs?
- Symbols resolved in order they are loaded
- Suppose our library containing `puts()` is `libhello.so`, located in a standard library directory (like `/usr/lib`), and we compile our `hello.c` code against this library:
  ```
  prompt% gcc -g -Wall hello.c -lhello -o hello
  ```
- Libraries specified using `-l` flag are loaded in order specified, and before C standard library
- Which `puts()` gets used here?
  ```
  athena% gcc -g -Wall hello.c -lc -lhello -o hello
  ```

Lecture 9

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Creating libraries

- Libraries contain C code like any other program
- Static or shared libraries compiled from (un-linked) object files created using `gcc`
- Compiling a static library:
  - compile, but do not link source files:
    `prompt% gcc -g -Wall -c infile.c -o outfile.o`
  - collect compiled (unlinked) files into an archive:
    `athena% ar -rcs libname.a outfile1.o outfile2.o ...`

Creating shared libraries

- Compile and do not link files using `gcc`:
  `prompt% gcc -g -Wall -fPIC -c` *`infile.c`* `-o` *`outfile.o`*
- `-fPIC` option: create position-independent code, since code will be repositioned during loading
- Link files using `ld` to create a shared object (`.so`) file:
  `prompt% ld -shared -soname lib`*`name`*`.so -o lib`*`name`*`.so.`*`version`* `-lc` *`outfile1.o outfile2.o ...`*
- If necessary, add directory to `LD_LIBRARY_PATH` environment variable, so `ld.so` can find file when loading at run-time
- Configure `ld.so` for new (or changed) library:
  `prompt% ldconfig -v`

Lecture 9

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

6.087 Practical Programming in C

January (IAP) 2010