Announcements

- **Program #3**
  - Due Tuesday (March 27) 8:00 PM
- **Reading**
  - Chapter 17 (Today)
    - skip 17.5.4
  - Chapter 13 (Tuesday)

Creating your own libraries

- **Libraries**
  - collections of object (.o) files that work together
  - can be linked into other programs
    - linking can happen prior to execution
    - linking can be done during program execution
- **Static libraries**
  - linked into the program as part of the link step
  - require space in each executable that uses them
- **Dynamic (shared) libraries**
  - linked into the program at program startup
  - require only one copy for the entire system
  - Have version numbers associated with them
    - controls which version work with which applications
Creating Static Libraries

- **UNIX Command ar**
  - LIBRARY=myLib.a
  - OBJS=file1.o file2.o file3.o
  - ar cru $(LIBRARY) $(OBJS)

- **Using a Static Library**
  - gcc -o myProgram main.o myLib.a

Creating Dynamic (shared) Libraries

- **Use special gcc flags**
  - -nostdlib -shared -fPIC -Wl,-soname,mylib.so.1
    - -nostdlib - no standard C library needed
    - -shared - generate a shared library
    - -fPIC - generate position independent code
    - -Wl,-soname,mylib.so.1
    - name the shared object mylib.so.1

- **Example Makefile Rules**

  LIBFLAGS = -nostdlib -shared -fPIC -Wl,-soname,$@.1

  libavl.so: avl.c table.h
  $(CC) $(LIBFLAGS) $(CFLAGS) avl.c -o libavl.so
  ln -f -s libavl.so libavl.so.1
Special Features of Shared Libraries

- **One option**
  - `void __init(void);`
    - Special function in shared libraries
    - Invoked automatically when the library is loaded
      - Works with pre-linked (-llibName)
      - Works when using dlopen
    - Allows library to run initialization code
      - Prepare internal data structures
      - Invoke functions to register actions with main program
  - `void __fini(void);`
    - Associated with dlclose

- **Another option**
  - `void __attribute__((constructor)) my_init(void);`
  - `void __attribute__((destructor)) my_fini(void);`
  - More recent addition to the library

Runtime Linking

- **Can Load a library at runtime**
  - Allows skins, plugins, etc to work

- **C functions to support this:**
  - `void *dlopen(const char *pathname, int mode)`
    - Pathname is the name of a shared library
    - Mode controls operation (RTLD_NOW)
  - `void *dlsym(void *handle, const char *name);`
    - Lookup a function by name in the passed shared library
    - Return a pointer to that function (or NULL if not found)
  - `int dlclose(void *handle);`
    - Returns 0 on success
Dlopen Example

typedef void (*displayFuncPtr)(char *file);
typedef struct {
    displayFuncPtr displayFunc;   char *mediaType;
} renderTypePlugin;

renderTypePlugin *loadPlugin(char *library, char *mediaType) {
    void *dlPtr;    renderTypePlugin *plugin;
    dlPtr = dlopen(library, RTLD_NOW);
    if (!dlPtr) return NULL;
    /* … allocate plugin and check return …*/
    plugin->displayFunc = (displayFuncPtr ) dlsym(dlPtr, "draw");
    if (!plugin->displayFunc)    /* error, couldn’t find function */
        plugin->mediaType = strdup(mediaType);
    dlclose(dlPtr);
    return plugin;
}

Implementation Hiding

- **Goals**
  - Export a public API
  - Hide details of implementation
    - allows implementation to evolve
    - better able to reason about a large system
      - don’t need to know how it works, just that it works
    - protects IP in implementation
  - Get the right abstractions
    - This is the hardest part!

- **Techniques**
  - No code in header files
  - Careful pointer definitions
    - opaque pointers
Opaque Pointer Types

- **Generic Pointers**
  - make API return void * pointers
  - useful for building containers
    - such as maps, sets, etc.
  - lacks type checking
  - Example:
    - void *createTable();
    - addItem(void *table, void *data);

- **Pointers to structs (struct definitions are hidden)**
  - Public header files define API and pointer to abstract types
    - API users only see public header files
  - Private header files define actual structures
    - Include public header files
    - Visible to modules implementing the API

Information Hiding Example

- **Project #3 revisited**
- **Change machine.h (to be the public header)**
  ```c
  struct _instruction;
  typedef struct _instruction Instruction;
  ```
- **Create machineP.h (private header)**
  ```c
  struct _instruction {
    unsigned int opCode:4;
    unsigned int r1:4;
    unsigned int r2:4;
    unsigned int r3:4;
    unsigned int address:16;
  };
  ```
Containers vs. APIs

- **Containers**
  - designed to hold a collection of the same or similar objects
  - provide common access to stored items

- **API**
  - encapsulates a useful bit of functionality
  - May use containers to store information

Common Container Abstractions

- **Stack**
  - Last in first out semantics
  - Push and pop are operations

- **Queue**
  - First in first out semantics
  - Enqueue and dequeue are only operations normally

- **Table (really a variation on a map)**
  - Storage of keyword, values
  - Supports insert, lookup, delete
Table Container

- Can have many underlying implementations
  - Arrays
  - Linked list
  - Hash table (open or chained)
  - Trees (many different trees possible)
- Each implementation has performance attributes
  - How much space is required?
  - How efficient is insert vs. lookup vs. delete
- No single right implementation for purposes
  - A class project will explores two alternative implementations

Designing a Good API

- Why it matters
  - Can always change a bad implementation
  - Changing an API is impacts all users
- Elements of a good design
  - Can be easily used for many different purposes
  - Focus on key ideas, not how they work
    - Lookup/Store data NOT insert into tree
  - Logical units of work with each function call
  - Simple (and common) things should be easy
    - add convenience functions if needed
      - functions that can be expressed using others in API
- Elements of a bad design
  - Exposes aspects on implementation
  - Abstractions don't match common use pattern
A Bad API Example

typedef struct {
    float real;
    float imaginary;
} complexNumber;

complexNumber *createComplex();
float getRealPart(complexNumber *num);
float getImaginaryPart(complexNumber *num);
void setNumber(complexNumber *num, float real, float imaginary);

• Hides nothing
• Provides little (to no) value added beyond struct

More Elements of good API Design

• Consistent Return Values across functions
  – 0 is always success
  – negative is always failure
• Consistent parameters
  – use the same names for the same things in different functions
  – how copy semantics are handled
    • are strings copied when passed to API