Miscellaneous C-programming Issues
Content

Pointers to functions
- Function pointers
- Callback functions
- Arrays of functions pointers

External libraries
- Symbols and Linkage
- Static vs Dynamic Linkage
- Linking External Libraries
- Creating Libraries Data
Function pointers

• In some programming languages, functions are first class variables (can be passed to functions, returned from functions etc.).

• In C, function itself is not a variable. But it is possible to declare pointer to functions.

• Question: What are some scenarios where you want to pass pointers to functions?

• Declaration examples:
  - `int (*fp)( int ) /*notice the () */`
  - `int (*fp)(void*,void*)`

• Function pointers can be assigned, pass to and from functions, placed in arrays.
Callbacks

• Definition: Callback is a piece of running code passed to functions. In C, callbacks are implemented by passing function pointers.

Example:

```c
void qsort(void* arr, int num, int size,
           int(*fp)(void* pa, void* pb))
```

• `qsort()` function from the standard library can be sort an array of any datatype.
  - Question: How does it do that? callbacks.

• `qsort()` calls a function whenever a comparison needs to be done.

• The function takes two arguments and returns (<0,0,>0) depending on the relative order of the two items.
Callback: qsort example

/* array definition */
int arr[] = {10, 9, 8, 1, 2, 3, 5};

/* callback */
int asc(void* pa, void* pb)
{
    return(*(int*)pa - *(int*)pb);
}

/* callback */
int desc(void* pa, void* pb)
{
    return(*(int*)pb - *(int*)pa);
}

/* sort in ascending order */
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), asc);

/* sort in descending order */
Callback: Apply example

- Consider a linked list with nodes defined as follows:

```c
struct node {int data ;
    struct node* next ;
};
```

Also consider the function `apply` defined as follows:

```c
void apply(struct node* phead, void (* fp ) (void* , void* ), void* arg)
/* only fp has to be named */
{
    struct node* p=phead ;
    while ( p !=NULL) {
        fp(p,arg);  /* or (* fp )(p,arg) */
        p=p->next ;
    }
}
```
Callback: Apply example (cont.)

• Iterating:

/* called back function */
void print(void* p, void* arg)
{
    struct node* np = (struct node*)p;
    printf("%d ", np->data);
}

struct node* phead;
/* populate somewhere */
apply(phead, print, NULL);
Array of function pointers

• Example: Consider the case where different functions are called based on a value.

```c
enum TYPE {SQUARE, RECT, CIRCILE, POLYGON};
struct shape {float params[MAX];
    enum TYPE type;};
void draw(struct shape* ps){
    switch (ps->type){
    case SQUARE: draw_square(ps); break;
    case RECT: draw_rect(ps); break;
    ... }
}```
Array of function pointers ('ed)

The same can be done using an array of function pointers instead.

```c
void (*fp[4])(struct shape* ps)=
{&draw_square,&draw_rec,&draw_circle,&draw_poly};
typedef void(*fp)(struct shape* ps) drawfn;
drawfn fp[4] =
{&draw_square,&draw_rec,&draw_circle,&draw_poly};

void draw(struct shape* ps){
    (* fp[ps->type])(ps); /* call the correct function */
```
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
Functions and variables as symbols

• Consider the simple hello world program written below:

```c
#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`:

```
athena% 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:

  athena% 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 does not bother creating symbols for unused function prototypes (saves space)
• What happens when we link?
  
  ```
  athena% 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:

\texttt{athena\% nm hello}

• Output: (other default symbols)

\begin{verbatim}
0000000000400524 T main
000000000040062c R msg
U puts@@GLIBC_2.2.5
\end{verbatim}

• Addresses for \texttt{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?

  athena% gcc -Wall -static hello.o -o hello
  hello

• Now contains the symbol puts:

  00000000004014c0 W puts
  0000000000400304 T main
  000000000046cd04 R msg

• ’W’: linked to another defined symbol
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, you 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 `libnamespec.so` or `libnamespec.a`, use compiler flag `-lnamespec` 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
Loading shared libraries on demand

• In Linux, can load symbols from shared libraries on demand using functions in dlfcn.h

• Open a shared library for loading:
  
  ```
  void * dlopen(const char *file, int mode);
  ```

  Modes:
  
  ```
  RTLD_LAZY (lazy loading of library),
  RTLD_NOW (load now), RTLD_LOCAL,
  RTLD_GLOBAL
  ```
Loading shared libraries on demand

- Get the address of a symbol loaded from the library:

  ```c
  void* dlsym(void * handle,
               const char* symbol_name);
  ```

  handle from call to `dlopen`; returned address is pointer to variable or function identified by `symbol_name`

- Need to close shared library file handle after done with symbols in library:

  ```c
  int dlclose(void * handle);
  ```

- These functions are not part of C standard library; need to link against library `libdl`: `-ldl` compiler flag
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:
    athena% gcc -g -Wall -c infile.c -o outfile.o
  - collect compiled (unlink ed) files into an archive:
    athena% ar -rcs libname.a outfile1.o outfile2.o
Creating shared libraries

• Compile and do not link files using gcc:
  athena% 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:
  athena% ld -shared -soname libname.so -o libname.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:
  athena% ldconfig -v
Thank you