Generic Programming in C

Void *
This is where the real fun starts
There is too much coding
everywhere else! ¹

- Using `void *` and function pointers to write generic code
- Using libraries to reuse code without copying and recompiling
- Using plugins to get run-time overriding and more!

---

Zero
Is where the Real Fun starts.
There’s too much counting
 Everywhere else!
-Hafiz
Function Pointers

- In C, the name of a function is a pointer!

```c
int f1(int x); /* prototype */

int (*func)(int); /* pointer to a fn with an int arg and int return */
func = f1;
n = (*func)(5); /* same as f1(5) */
n = func(5); /* same as (*func)(5) */
```

- Dereferencing a function pointer just returns a pointer to the function, so we don’t need to dereference it before calling it (but we can if we want to).

- We can also have an array of function pointers.

```c
int (*bagOfTricks[10])(int, char *);
```

- The prototype for quicksort function `qsort` in the standard C library uses a function pointer to compare function to enable a generic sort function (man 3 qsort).

```c
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```
# Function Pointer Examples

- **Function Pointer Example 1**
  - C-examples/function-pointers/funkyfun.c
  - C-examples/function-pointers/funky.c
  - C-examples/function-pointers/funky.h

- **Function Pointer Example 2**
  - C-examples/function-pointers/test-qsort.c

- **Function Pointer Example 3**
  - C-examples/function-pointers/fun-with-fns.c

- **Function Pointer Example 4**
  - C-examples/objects/Address.h
  - C-examples/objects/Address.c
  - C-examples/objects/testAddress.c
We can add pointers to functions that operate on the linked list as members of the list structure itself.

```c
struct list {
    int size;
    struct node *head;
    struct node *tail;
    int (*equals)(const void *, const void *);
    char *(*toString)(const void *);
    void (*freeObject)(void *);
};
```

- `int (*equals)(const void *, const void *)`
  A pointer to a function that takes two generic pointers and returns an int.

- `char *(*toString)(const void *)`
  A pointer to a function that takes a generic pointer and returns a string.

- `void (*freeObject)(void *)`
  A pointer to a function that takes a generic pointer and returns nothing.
Function Pointers: Function Pointer Example 4 (2)

```c
struct list *list = (struct list *) malloc(sizeof(struct list));

list->toString = printItemShort; /* assign custom function */
list->toString(list); /* call toString function */

list->toString = printItemLong; /* assign different function */
list->toString(list); /* call toString function */
```
The type `void *` is used as a generic pointer in C (similar in concept to `Object` type in Java). Pointers to any type can be assigned to a variable of type `void *`, which allows **polymorphism**.

The following example shows a polymorphic min function that works for an array of any type as long as we have a compare function for two elements of the array.

```c
/* Find min in v[0..size-1], assumes size is > 0 */
int min(void v[], int size, int (*compare)(void *, void *))
{
    int i;
    int min = 0;
    for (i = 1; i < size; i++) {
        if (compare(v[i], v[min]) < 0) {
            min = i;
        }
    }
    return min;
}
```
Polymorphic Quicksort

```c
void swap(void *v[], int i, int j)
{
    void *tmp = v[i]; v[i] = v[j]; v[j] = tmp;
}

/* qsort : sort v[left]..v[right] into increasing order */
void qsort(void *v[], int left, int right,
           int (*compare)(void *, void *))
{
    int i, last;

    if (left >= right) /* do nothing if array contains */
        return; /* less than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i<= right; i++) {
        if (compare(v[i], v[left]) < 0) {
            swap(v, ++last, i);
        }
    }
    swap(v, left, last);
    qsort(v, left, last-1, compare);
    qsort(v, last+1, right, compare);
}
```
A library is a collection of code that implements commonly used methods or patterns with a public API. This is combined with generic code to facilitate code re-use.

Libraries can be shared (also known as dynamically linked libraries or DLLs) or be static.

Is a shared library part of the process or is it a resource? It should be viewed as a resource since the operating system has to find it on the fly.

A static library, on the other hand, becomes part of the program text.

The concept of re-entrant code, i.e., programs that cannot modify themselves while running. Re-entrant code is necessary to write libraries.
The standard C library is automatically linked into programs when you compile with gcc.

The shared version of the library can be found at /usr/lib64/libc.so (or /usr/lib/libc.so).

If the static libraries are installed on your system, the static version of the standard library can be found at /usr/lib64/libc.a (or /usr/lib/libc.a).

Check out other libraries in the lib directory.

Any other libraries need to be explicitly included and the linker needs to know where find them.
Library Conventions

- A library filename always starts with `lib`.
- File suffixes
  - `.a` : Static libraries
  - `.so` : Shared libraries
- The library must provide a header file that can be included in the source file so it knows of function prototypes, variables, etc.
Creating a Shared Library (Linux)

- Suppose we have three C files: `f1.c`, `f2.c`, and `f3.c` that we want to compile and add into a shared library that we will name `mylib`. First, we can compile the C files with the flags `-fPIC -shared` to the `gcc` compiler.
  ```
  gcc -I. -Wall -fPIC -shared -c -o f1.o f1.c
  gcc -I. -Wall -fPIC -shared -c -o f2.o f2.c
  gcc -I. -Wall -fPIC -shared -c -o f3.o f3.c
  ```

- Then we can combine, the three object files into one shared library using the `ld` linker/loader.
  ```
  ld -fPIC -shared -o libmylib.so f1.o f2.o f3.o
  ```

- Now we can compile a program that invokes functions from the library by linking it with the shared library.
  ```
  gcc -I. -L. test1.c -o test1 -lmylib
  ```
  The compiler will search for the shared library named `libmylib.so` in the current folder as well as a set of system library folders.

- If your shared library is in some other folder, you can specify that folder with the `-L` option. For example, if your library is in the sub-folder `lib` underneath the current folder, you can use
  ```
  gcc test1.c -o test1 -Llib -lmylib
  ```

- When you run the executable, again the system has to be able to find the shared library. If it is not in the current folder (or installed in a system folder), then use the environment variable `LD_LIBRARY_PATH` to specify what set of folders to search in. For example:
  ```
  export LD_LIBRARY_PATH=./:lib:$LD_LIBRARY_PATH
  ```
Creating a Static Library

Suppose we have three C files: f1.c, f2.c, and f3.c that we want to compile and add into a static library that we will name mylib. First, we will compile the C files with the flag -fPIC to the gcc compiler.

```bash
gcc -I. -Wall -fPIC -c -o f1.o f1.c
gcc -I. -Wall -fPIC -c -o f2.o f2.c
gcc -I. -Wall -fPIC -c -o f3.o f3.c
```

Then we can combine, the three object files into one static library using the ar archive program.

```bash
ar rcv libmylib.a f1.o f2.o f3.o
```

At this point, we can write a test program that invokes functions from the library and link it with the static library.

```bash
gcc -I. -Wall -static -L. test1.c -lmylib -o test1.static
```

The rules for finding a static library are the same as for shared libraries. Note that for the above command to work, you will need to have a static version of the standard C library. You can install that with the command (on Fedora Linux):

```
yum install glibc-static
```
Shared libraries: dynamic versus static linking

- **Dynamic linking.** This is the default. Here the library is kept separate from the executable, which allows the library code to be shared by more than one executable (or more than one copy of the same executable). Makes the executable size smaller leading to faster loading up time. The downside is that if the we move the executable to another system but the library is missing there, the executable will not run!

- **Static linking.** With static linking we can create a self contained executable. We need to create a static library that we can then link to using the `-static` option with the C compiler. See the output below. Note that static executable are very large in comparison to dynamically linked executables.

```
[amit@dslamit libraries]$ ls -l
-rw-r-xr-x 1 amit home 2503 Aug 30 00:16 libmylib.so
-rw-r-xr-x 1 amit home 5220 Aug 30 00:16 test1
-rw-r-xr-x 1 amit home 404507 Aug 30 00:16 test1.static
```
We can use libraries to help organize our own projects as well. For example, we can create a library that contains our list, binary search tree and hash table classes. Then we can link to that library to use in our projects instead of having to copy the code.

We can build a library and simply copy the library into the same folder as our executable program. The system will find the library since it is in the same folder.

Create a **lib** folder to hold your libraries, an **include** folder for the header files inside your project. With this approach, your project is self-contained although you do have to copy the library into each project.

Install the library into one of the standard system library folders like **/lib**, **/usr/lib**, **/usr/local/lib** etc. Install the header files in a standard system header file folder like **/usr/include**, **/usr/local/include** etc.
Assuming that we have installed our libraries in $HOME/lib and the header files in $HOME/include.

When a program starts, the system will search for libraries using the environment variable name LD_LIBRARY_PATH. We can add the following line to the end of the file ~/.bashrc.

```bash
export LD_LIBRARY_PATH=$HOME/lib:$LD_LIBRARY_PATH
```

You will also have to add the option -L$HOME/lib when you compile your programs so that the compiler will search for libraries in the subfolder named lib in the current folder.
Similarly, we can place all the relevant header files (like `Job.h`, `Node.h`, and `List.h`) in the directory `include` in your project folder. Then include the option `-I$HOME/include` anytime you are compiling a program that uses the List class. You can now include the header files as follows:

```c
#include <Job.h>
#include <Node.h>
#include <List.h>
```

The compiler will automatically search for the files in the `include` directory in your current folder.

After the above two steps, you can simply link in your library without having to copy the code to each new directory you may want to work in.

```bash
gcc -Wall -I$HOME/include -L$HOME/lib -o prog1 program1.c -lmylib
```
How to check for library dependency?

- **Linux**: Use the tool `ldd`.
- **MS Windows**: Use the tool `depends` (available from [http://www.dependencywalker.com](http://www.dependencywalker.com)).
- **MacOSX**: Use the tool `otool`.
A plugin is a piece of code that can be loaded into or unloaded from a program upon demand without having to restart the program.

For C and C++ programs plugins are typically implemented as shared libraries that are dynamically loaded (and sometimes unloaded) by the main program.

Example applications include media players (e.g., an MPEG4 player) for web browsers, specialized filters for image processing applications (e.g., Adobe Photoshop), various plugins to extend the functionality of Eclipse, dynamically loadable drivers for operating systems (e.g., Linux modules) and so forth.
Using plugins requires support from the operating system. Under Linux, the system calls `dlopen`, `dlsym`, `dlclose`, `dlerror` provide the programming interface to dynamic linking loader. These calls are Linux specific (and not ANSI C).

```c
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);
const char *dlerror(void);
void *dlsym(void *handle, char *symbol);
int dlclose(void *handle);
```

There are man pages for each of these system calls.

Equivalent versions are available under other operating systems.
void * libHandle = dlopen("/opt/lib/libmylib.so", RTLD_LAZY);

Open shared library named "libmylib.so". The second argument indicates the binding. Returns NULL if it fails.

Options:

- RTLD_LAZY: If specified, Linux is not concerned about unresolved symbols until they are referenced. Commonly used option.
- RTLD_NOW: All unresolved symbols must be resolved when dlopen() is called.
- RTLD_GLOBAL: Make symbol libraries visible.
Plugins in Linux (3)

dlsym(libHandle, "myfunc");

Returns address to the function which has been loaded with the shared library. Returns NULL if it fails. Options:

- **dlsym()** allows a process to obtain the address of a symbol defined within an library made accessible through a `dlopen()` call.
- **libHandle** is the value returned from a call to `dlopen()` (and which has not since been released via a call to `dlclose()`)
- **myfunc** is the function’s (symbo’s) name as a character string. `dlsym()` will search for the named symbol in all libraries loaded automatically as a result of loading the library referenced by the handle.
Plugin: Example 1 (1)

/* plugin1.c */
* gcc -Wall -c plugin1.c
* gcc -shared -lc -o plugin1.so plugin1.o

#include <stdio.h>
void plugin(void)
{
    printf("This is plug-in 1\n");
}
Plugin: Example 1 (2)

```c
/* plugin2.c */
* gcc -Wall -c plugin2.c
* gcc -shared -lc -o plugin2.so plugin2.o
* /

/* C-examples/plugin/plugin2.c */
#include <stdio.h>

void plugin(void)
{
    printf("This is the second plug-in\n");
}
```
/* C-examples/plugin/runplug.c */
#define MAX_BUF 1024
#include <stdio.h>
#include <string.h>
#include <dlfcn.h>

void *handle; /* handle of shared library */
void (*function)(void); /* pointer to the plug-in function */
const char *dlError; /* error string */

int main(int argc, char **argv) {
    char buf[MAX_BUF];
    char plugName[MAX_BUF];

    while (1) {
        /* get plug-in name */
        printf("Enter plugin name (exit to exit): ");
        fgets(buf, MAX_BUF, stdin);
        buf[strlen(buf)-1] = '\0'; /* change \n to \0 */
        sprintf(plugName, "./%s", buf); /* start from current dir */

        /* checks for exit */
        if (!strcmp(plugName, "./exit"))
            return 0;
    }
}
Plugin: Example 1 (4)

/* C-examples/plugin/runplug.c */

...

/* open a library */
handle = dlopen(plugName, RTLD_LAZY);
if ((dlError = dlerror())) {
    printf("Opening Error: %s\n", dlError);
    continue;
}

/* loads the plugin function */
function = dlsym(handle, "plugin");
if ((dlError = dlerror()))
    printf("Loading Error: %s\n", dlError);

/* execute the function */
(*function)();
if ((dlError = dlerror()))
    printf("Execution Error: %s\n", dlError);

/* close library 1 */
dlclose(handle);
if ((dlError = dlerror()))
    printf("Closing Error: %s\n", dlError);
}
return 0;
More on plugins

▸ Create an API for the plugins for your program and publicize it!

▸ Then others can write plugins for your program and thus extend its functionality... :-)

▸ Users can install the plugins that they want rather than have a large program with all possible function in it.