Generic Programming in C

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

- Variable argument lists
- 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!

---

1 *Zero*
Is where the Real Fun starts.
There's too much counting
Everywhere else!
-Hafiz

Variable Argument Lists in C (1)

- C allows a function call to have a variable number of arguments with the variable argument list mechanism.
- Use *ellipsis* `...` to denote a variable number of arguments to the compiler. the ellipsis can only occur at the end of an argument list.
- Here are some standard function calls that use variable argument lists.

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
int execlp(const char *file, const char *arg, ...);
```

- See `man stdarg` for documentation on using variable argument lists. In particular, the header file contains a set of macros that define how to step through the argument list.
- See Section 7.3 in the K&R C book.

Variable Argument Lists in C (2)

Useful macros from `stdarg` header file.

- `va_list argptr;` is used to declare a variable that will refer to each argument in turn.
- `void va_start(va_list argptr, last);` must be called once before `argptr` can be used. `last` is the name of the last variable before the variable argument list.
- `type va_arg(va_list ap, type);` Each call of `va_arg` returns one argument and steps ap to the next; `va_arg` uses a type name to determine what type to return and how big a step to take.
- `void va_end(va_list ap);` Must be called before program returns. Does whatever cleanup is necessary.
- It is possible to walk through the variable arguments more than once by calling `va_start` after `va_end`.

Variable Argument Lists Example

```
/* C-examples/varargs/test-varargs.c */
#include <stdio.h>
#include <stdarg.h>

void strlist(int n, ...)
{
  va_list ap;
  char *s;

  va_start(ap, n);
  while (1) {
    s = va_arg(ap, char *);
    printf("%s\n",s);
    n--;
    if (n==0) break;
  }
  va_end(ap);
}
int main()
{
  strlist(3, "string1", "string2", "string3");
  strlist(2, "string1", "string3");
}
```

Function Pointers

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

```
int f1(int x); /* prototype */
/* pointer to a fn with an int arg and int return */
int (*func)(int);

func = f1;
n = (*func)(5); /* same as f1(5)
```

- We can also have an array of function pointers.

```
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.

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

test-qsort.c: Function Pointer Example 1 (1)

```
/* C-examples/function-pointers/test-qsort.c */
#include <stdlib.h>
#include <stdio.h>

/* qsort needs a compare function that returns
     0 if x '==' y
    <0 if x '<' y
    >0 if x '>' y
*/
int compareInt(const void *x, const void *y)
{
        return ((*(int *)x) - (*(int *)y));
}
struct student {
    int id;
    char *name;
    char *address;
};

int compareId(const void *x, const void *y)
{
        int key1, key2;
        key1 = ((struct student *)x)->id;
        key2 = ((struct student *)y)->id;
        return (key1 - key2);
}
```

test-qsort.c: Function Pointer Example 1 (2)

```
int main(int argc, char **argv)
{
    int i, n;
    int *array;
    char *strings;
    if (argc != 2) {
        fprintf(stderr, "Usage:  %s  <n>\n",  argv[0]) ;
        exit(1);
    }
    n = atoi(argv[1]);
    array = (int *) malloc(sizeof(int)*n);
    srandom(0);
    for (i=0; i<n; i++) {
        array[i] = random() % n;
    }
    qsort(array, n, sizeof(int), compareInt);

    roster = (struct student *) malloc(sizeof(struct student)*n);
    for (i=0; i<n; i++) {
        roster[i].id = n-i;
        roster[i].name = NULL;
        roster[i].address = NULL;
    }
    qsort(roster, n, sizeof(struct student), compareId);

    exit(0);
}
```

fun-with-fns.c: Function Pointer Example 2 (1)

```c
/* C-examples/function-pointers/fun-with-fns.c */
#include <stdio.h>
#include <stdlib.h>

int foobar0(int x) {
    printf("I have been invoked!!!! x=%d\n",x);
    return x;
}

int foobar1(int x) {
    printf("I have been invoked!!!! x=%d\n",x*2);
    return x;
}

int foobar2(int x) {
    printf("I have been invoked!!!! x=%d\n",x*3);
    return x;
}
void fun(int (*fn)(int x)) {
    int result;

    result = (*fn) (5);
}
```

fun-with-fns.c: Function Pointer Example 2 (2)

```
int main(int argc, char **argv)
{
    int i;
    int count=0;
    int (*names[3])(int);

    names[0] = foobar0;
    names[1] = foobar1;
    names[2] = foobar2;

    if (argc != 2) {
        fprintf(stderr,"Usage %s: <count>\n",argv[0]);
        exit(1);
    }
    count = atoi(argv[1]);
    for (i=0; i<count; i++) {
        fun(names[random()%3]);
    }
    exit(0);
}
```

objects/Address.h: Function Pointer Example 3 (1)

```
/* C-examples/objects/Address.h */
#ifndef ADDRESS_H
#define ADDRESS_H

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef struct address address;
typedef struct address * Address;

struct address {
    char *name;
    char *streetAddress;
    char *city;
    char *state;
    int zip;

    char * (*toString)(Address);
};

Address createAddress(char *, char *, char *, char *, int);
char *toString(Address);
char *printAddress(Address);

#endif /* ADDRESS_H */
```

objects/Address.c: Function Pointer Example 3 (2)

```
/* C-examples/objects/Address.c */
#include "Address.h"

static int getLength(Address this)
{
    return strlen(this->name)+strlen(this->streetAddress)+
           strlen(this->city)+strlen(this->state)+4+5+10;
}

Address createAddress(char *name, char *streetAddress, char *city,
                      char *state, int zip)
{
    Address temp = (Address) malloc(sizeof(address));
    temp->name = (char *) malloc(sizeof(char)*(strlen(name)+1));
    temp->streetAddress = (char *)
    malloc(sizeof(char)*(strlen(streetAddress)+1));
    temp->city = (char *) malloc(sizeof(char)*(strlen(city)+1));
    temp->state = (char *) malloc(sizeof(char)*(strlen(state)+1));

    strcpy(temp->name, name);
    strcpy(temp->streetAddress, streetAddress);
    strcpy(temp->city, city);
    strcpy(temp->state, state);
    temp->zip = zip;

    temp->toString = toString;
    return temp;
}
```

objects/Address.c: Function Pointer Example 3 (3)

```
char *toString(Address this)
{
    int len = getLength(this);
    char * temp = (char *) malloc(sizeof(char)*len);
    snprintf(temp, len, "%s\n%s\n%s, %s, %d\n", this->name,
     this->streetAddress, this->city, this->state, this->zip);
    return temp;
}

char *printAddress(Address this)
{
    int len = getLength(this);
    char * temp = (char *) malloc(sizeof(char)*len);
    snprintf(temp, len, "%s, %s, %s, %s %d\n", this->name,
     this->streetAddress, this->city, this->state, this->zip);
    return temp;
}
```

objects/testAddress.c: Function Pointer Example 3 (4)

```
/* C-examples/objects/testAddress.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "Address.h"

int main(int argc, char *argv[])
{
    Address addr1 = createAddress("Moo Shoo", "123 Main Street",
                                  "Boise", "Idaho", 83701);
    printf("%s\n", (*addr1->toString)(addr1));
    addr1->toString = printAddress;
    printf("%s\n", (*addr1->toString)(addr1));
    exit(0);
}
```

Function Pointers: Function Pointer Example 4 (1)

We can store pointers to functions that operate on the *List* in the `List` structure itself!

```
typedef struct list List;
typedef struct list * ListPtr;
struct list {
    int size;
    ListPtr head;
    ListPtr tail;

    ListPtr (*createList)();
    void (*addAtFront)(ListPtr list, NodePtr node);
    NodePtr (*removeFront)(ListPtr list);
};
```

Function Pointers: Function Pointer Example 4 (2)

```
ListPtr myList = (ListPtr) malloc(sizeof(List));
/* use my custom functions */
myList->createList = myCreate;
myList->addAtFront = myAddAtFront;
myList->removeFront = myRemoveFront;


/* example of using functions */
myList->(*createList)();
/* assume node has been created appropriately */
myList->(*addAtFront)(list, node);


/*  Now...I want a different addAtFront function */
myList->addAtFront = anotherAddAtFront;
/* below we use the new addAtFront function */
myList->(*addAtFront)(list, node);
```

Generic Programming

- 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.

```
/* 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

```
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 Generic Doubly-Linked List

- In order to create a generic doubly-linked list, we have a List class that uses a Node class but now each node contains a `void *` pointer to a generic object that will be stored in our list.
- We also need for the user to pass us three function pointers: one for obtaining the key value for a given object, another for freeing the object and also one for converting the object into a string representation suitable for printing. We will store these function pointers in the List structure.

```
typedef struct list List;
typedef struct list *ListPtr;
struct list {
    int size;
    NodePtr head;
    NodePtr tail;
int (*compareTo)(const void *, const void *);
    char *  (*toString)(void *);
    void (*freeObject)(void *);
};
/* constructor */
ListPtr createList(int (*compareTo)(const void *, const void *);
                   char * (*toString)(void *),
                   void (*freeObject)(void *));
/* other function prototypes are unchanged */
```

- We can make it “more generic" by using a compare function pointer instead of a function to get a key. That is left as an exercise for you!

Node.h

```
#ifndef __NODE_H
#define __NODE_H
/* C-examples/doublyLinkedLists/library/Node.h */

#include <stdio.h>
#include <stdlib.h>
#include "common.h"

typedef struct node Node;
typedef struct node *NodePtr;

struct node {
    void *obj;
    NodePtr next;
    NodePtr prev;
};

NodePtr createNode (void *obj);
void freeNode(NodePtr node, void (*freeObject)(void *));
#endif /* __NODE_H */
```

List.c

```
/* C-examples/doublyLinkedLists/library/List.c */

ListPtr createList(int (*compareTo)(const void *, const void *);
                   char * (*toString)(void *),
                   void (*freeObject)(void *))
{
    ListPtr list;
    list = (ListPtr) malloc(sizeof(List));
    list->size = 0;
    list->head = NULL;
    list->tail = NULL;
    list->compareTo = compareTo;
    list->toString = toString;
    list->freeObject = freeObject;
    return list;
}

/* other functions unchanged from earlier example */
```

Node.c

```
/* C-examples/doublyLinkedLists/library/Node.c */
#include "Node.h"

NodePtr createNode(void *obj)
{
    NodePtr newNode = (NodePtr) malloc (sizeof(Node));
    newNode->next = NULL;
    newNode->prev = NULL;
    newNode->obj = obj;
    return newNode;
}
void freeNode (NodePtr node, void (*freeObject)(void *))
{
    if (node == NULL) return;
    (*freeObject)(node->obj);
    free(node);
}
```

Using the Generic List

- In order to use the generic list, the user will have to create an object type that they want to use (can be any type with any name).
- Then they have to provide the three function pointers for `compareTo`, `toString` and `freeObject` to the `createList` function in the List class.
- See the example in the following directories: `C-examples/doublyLinkedLists/generic/`
- **Recommended Exercise**. Go convert your previous project on doubly linked lists into a generic version!

Trees

- **Binary Tree**. To declare a binary tree, we can use something like the following declaration.

```
typedef struct TreeNode TreeNode;
typedef struct TreeNode *TreeNodePtr;

struct TreeNode {
    int key;
    void *data;
    TreeNodePtr left;
    TreeNodePtr right;
}
```

- **M-ary Tree**. Here is an example of a tree where each node can have up to `MAX_DEGREE=M` child nodes.

```
typedef struct TreeNode TreeNode;
typedef struct TreeNode *TreeNodePtr;

struct TreeNode {
    int key;
    void *data;
    TreeNodePtr child[MAX_DEGREE];
}
```

Recommended Exercise

- Develop a complete header file for a "generic" Binary Search Tree class in C. What operations would you want to provide? How would you store data in the Binary Search Tree? We will at least need methods for searching, inserting, deleting, inorder traversal among others.

Libraries

- 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.

Standard C Libraries

- The standard C library is automatically linked into pgrograms 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 verision 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.

```
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.

```
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.

```
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
-rwxr-xr-x  1 amit home   2503 Aug 30 00:16 libmylib.so
-rwxr-xr-x  1 amit home   5220 Aug 30 00:16 test1
-rwxr-xr-x  1 amit home 404507 Aug 30 00:16 test1.static
```

Using Shared Libraries in Linux (1)

- 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.

Using Shared Libraries in Linux (2)

- 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`.
  ```
  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.

Using Shared Libraries in Linux (3)

- Similarly, we can place all the relevant header files (like `Job.h`, `Node.h`, and `List.h`) in the 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:

  ```
  #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.

```
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`).
- **MacOSX**: Use the tool otool.

Plugins

- 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.

Plugins in Linux (1)

- 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).

```
#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.

Plugins in Linux (2)

```
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

/* C-examples/plugin/plugin1.c */

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

Plugin: Example 1 (2)

```
/* 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");
}
```

Plugin: Example 1 (3)

```
/* 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 funcionality in it.