

Using C for CGI Programming

Clay Dowling

February 27, 2004

Perl, Python and PHP are the holy trinity of CGI application programming. Stores have shelves full of books on these topics, they're covered well in the computer press, and there's plenty on the Internet about them. There is a distinct lack of information on using C to write CGI applications. In this article I will show you how to use C for CGI programming, and lay out some situations where it will provide significant advantages.

I use C in my applications for three reasons: speed, features and stability.

Although the conventional wisdom is against it, my own benchmarks have found that C and PHP are equivalent in speed when the processing to be done is simple, and C will win hands-down when there is any complexity to the processing. I have not run benchmarks with Perl and Python, so I can't speak to speed differences there.

C provides an excellent feature set as well. The language comes with a bare-bones set of features, but there is a staggering number of libraries available for nearly any purpose that a computer can be put to. Perl of course is no slouch in this area, and I won't contend that C offers more extensibility, but both can fill nearly any bill.

CGI programs written in C are very stable. Because the program is compiled, it is not as susceptible to changes in the operating environment as PHP is. Also, because the language is stable and well developed, it does not experience the dramatic changes that PHP users have been subjected to over the last few years.

1 The Application

My application will be a simple event listing. It's suitable for a business to list upcoming events (e.g. the meeting schedule for a day, or events at a church). It will provide an administrative interface intended to be password protected, and a public interface that lists all upcoming events (but only upcoming events). I'll provide for runtime configuration and interface independence.

Data store will be in a database, rather than writing my own data store. There will be an configuration file that contains the database connection information. There will also be a collection of files used to provide interface/code separation.

The administrative interface will allow events to be listed, edited, saved and deleted. Listing events will be the default action, if no other action is provided. Both new and existing events can be saved. The interface consists of a grid screen which displays the list of events, and a detail screen which contains the full record of a single event.

Figure 1: MySQL Schema

```
CREATE TABLE event (  
  event_no int(11) NOT NULL auto_increment,  
  event_begin date NOT NULL default '0000-00-00',  
  name varchar(80) NOT NULL default '',  
  location varchar(80) NOT NULL default '',  
  begin_hour varchar(10) default NULL,  
  end_hour varchar(10) default NULL,  
  event_end date NOT NULL default '0000-00-00',  
  PRIMARY KEY (event_no),  
  KEY event_date (event_begin)  
)
```

The database schema for this application consists of a single table, defined in listing 1. This schema is MySQL specific, but an equivalent schema can be created with any database engine.

The following functions will be minimally necessary to implement the functionality of the administrative interface: `list_events()`, `show_event()`, `save_event()` and `delete_event()`.

I am also going to abstract the reading and writing of database data into their own group of functions. This keeps each function simpler, which makes debugging easier. The functions that I'll need for the data storage interface are: `event_create()`, `event_destroy()`, `event_read()`, `event_write()` and `event_delete()`. To make my life easier, I'm also going to add `event_fetch_range()` so that I can choose a range of events, since I need to do this in at least two places.

I'll abstract my records to C structures. I'll also abstract database result sets to linked lists. Abstraction lets me change database engines or data representation with relatively little expense, because only a little part of my code deals directly with the data store.

There isn't room to print all of my source code in this article. Complete source code and my Makefile can be downloaded from my web site (<http://www.lazarusid.com/eventcalendar/>)

2 Tools

The first hurdle to overcome when using C is the set of tools that you will use. At bare minimum you will need a CGI parser to break out the CGI information for you. Chances are very good that you're also looking for some database connectivity. A little bit of logic/interface independence is good too, so that you aren't re-writing code every time the site needs a makeover.

For CGI parsing I recommend the cgic library from Thomas Boutell (<http://www.boutell.com/cgic>). It's shockingly easy to use and provides access to all parts of the CGI interface. If you're a C++ person, the cgicc libraries (<http://www.cgicc.org>) are also suitable, although I found the Boutell library to be easier to use.

Since MySQL is pretty much the standard for UNIX web development, I'll stick with it for my sample application. Every significant database engine has a functional C interface library though, so you can use whatever database you like.

I'm going to provide my own interface independence routines, but you could use libxml and libxslt to do the same thing with a good deal more sophistication.

3 Runtime Configuration

At runtime, I need to be able to configure the database connection. Given a filename and an array of character strings for the configuration keys, my configuration function will populate a corresponding array of configuration values.

Now I can populate a string array with whatever keys I've chosen to use and get the results back in the value array.

4 User Interface

There are two parts to the user interface. As a programmer I'm primarily concerned with the input forms and URL strings. Everybody else will care how the page around my form looks, and just take the form for granted. The solution to keep both parties happy is to have the page exist separately from the form and my program.

Templating libraries abound in PHP and Perl, but there are no common HTML templating libraries in C. The easiest solution is to include only the barest minimum of the output in my C code, and keep the rest in html files which are output at the appropriate time. A function which will do this for us is found in listing 3.

Before generating output, I need to tell the web server and the browser what I'm sending. `cgiHeaderContentType()` will do this for us. I want a content type of "text/html", so I pass that as the argument. The general steps to follow for any page I want to display are:

- `cgiHeaderContentType("text/html");`
- `html_get(path, pagetop.html);`
- Generate the program content
- `html_get(path, pagebottom.html);`

5 Form Processing

Now that I can generate a page and print a form, I need to be able to process that form. I need to read both numeric and text elements, so I'll use a couple of functions from the

Figure 2: Runtime configuration function

```
void config_read(char* filename, char** key,
                 char** value) {

    FILE* cfile;
    char tok[80];
    char line[2048];
    char* target;
    int i;
    int length;

    cfile = fopen(filename, "r");
    if (!cfile) {
        perror("config_read");
        return;
    }

    while(fgets(line, 2048, cfile)) {
        if ((target = strchr(line, '='))) {
            sscanf(line, "%80s", tok);
            for(i=0; key[i]; i++) {
                if (strcmp(key[i], tok) == 0) {
                    target++;
                    while(isspace(*target)) target++;
                    length = strlen(target);
                    value[i] = (char*)calloc(1, length + 1);
                    strcpy(value[i], target);
                    target = &value[i][length - 1];
                    while(isspace(*target)) *target-- = 0;
                }
            }
        }
    }
    fclose(cfile);
}
```

Figure 3: HTML template function

```
void html_get(char* path, char* file) {  
  
    struct stat sb;  
    FILE* html;  
    char* buffer;  
    char fullpath[1024];  
  
    /* File & path name exceed system limits */  
    if (strlen(path) + strlen(file) > 1024) return;  
  
    sprintf(fullpath, "%s/%s", path, file);  
    if (stat(fullpath, &sb)) return;  
  
    buffer = (char*)calloc(1, sb.st_size + 1);  
    if (!buffer) return;  
    html = fopen(fullpath, "r");  
    fread((void*)buffer, 1, sb.st_size, html);  
    fclose(html);  
    puts(buffer);  
    free(buffer);  
  
}
```

cgic library: `cgiFormStringNoNewlines()` and `cgiFormInteger()`. The cgic library implements the main function for me, and requires that I implement `int cgiMain(void)`. `cgiMain()` is where I put the bulk of my form processing.

To display a single record in my `show_event` function, I get the `event_no` (my primary key) from the cgi parameter `eventno`. `cgiFormInteger()` will retrieve an integer value, and set a default value if no cgi parameter was provided.

I'll also need to get a whole raft of data from the form in `save_event`. Dates are a thorny thing to input because they consist of three pieces of data: year, month and date. I need both a begin and end date, which gives me six fields to interpret. I also need to input the name of the event, begin and end times (which are strings, because they might be events themselves, such as sunrise or sunset) and the location. Listing 4 shows how this works in code.

Listing 4 also demonstrates `cgiHeaderLocation()`. This function redirects the user to a new page. After I've saved the submitted data, I want to show the event listing page. Instead of a literal string, I have used one of the variables that libcgic provides: `cgiScriptName`. Using this variable instead of a literal means that the program name can be changed without breaking the program.

Finally, I need a way to handle my submit buttons. They're my most complex input, because I need to launch a function based on their value, and select a default value just in case. The cgic library has a function `cgiFormSelectSingle()` which emulates this behavior exactly. It requires the list of possible values to be in an array of strings, and populates an integer variable with the index of the parameter in the array or a default value if there were no matches.

I found <http://www.functionpointers.org> to be a useful resource for coming up to speed on function pointers. If function pointers still give you fits, you can choose the function to run in a switch statement. I prefer the array of function pointers because it is more compact, but my older code still makes use of the switch statement.

6 Database System

MySQL from C is largely the same as PHP, if you're used to that interface. You'll have to use MySQL's string escape functions to escape problematic characters in your strings such as quote characters or the backslash character, but otherwise you'll find it pretty much the same.

The `show_event()` function requires me to fetch a single record from the primary key. All of the error checking bulks up the code, but it's really three basic statements.

A call to `mysql_query()` executes my SQL statement and generates a result set. A call to `mysql_store_result()` retrieves the result set from the server. Finally, a call to `mysql_fetch_row()` gets a single `MYSQL_ROW` variable out of the result set.

The `MYSQL_ROW` variable can be treated like an array of strings (`char**`). If any of the data is numeric, and you wish to treat it as numeric data, you will need to convert it. For instance, in my application it is desirable to have the date as three separate numeric components. Because this data is structured as YYYY-MM-DD, I'll use `sscanf()` to get the components.

Figure 4: save_event(), parsing CGI data

```
struct event* e;

e = event_create();
cgiFormInteger("eventno", &e->event_no, 0);
cgiFormStringNoNewlines("name", e->name, 80);
cgiFormStringNoNewlines("location",
                        e->location, 80);

/* Processing date fields */
cgiFormInteger("beginyear",
              &e->event_begin->year, 0);
cgiFormInteger("beginmonth",
              &e->event_begin->month, 0);
cgiFormInteger("beginday", &e->event_begin->day, 0);
cgiFormInteger("endyear", &e->event_end->year, 0);
cgiFormInteger("endmonth", &e->event_end->month, 0);
cgiFormInteger("endday", &e->event_end->day, 0);

/* Process begin & end times separately */
cgiFormStringNoNewlines("beginhour",
                      e->event_begin->hour, 10);
cgiFormStringNoNewlines("endhour",
                      e->event_end->hour, 10);

event_write(e);

cgiHeaderLocation(cgiScriptName);
```

Figure 5: Handling submit buttons

```
char* command[5] = {"List", "Show", "Save", "Delete", 0};
void (*action)(void)[5] = {list_events, show_event,
                          save_event, delete_event, 0};
int result;

cgiFormSelectSingle("do", command, 4, &result, 0);
action[result]();
```

Figure 6: Retrieving data from MySQL

```
MYSQL_RES* res;
MYSQL_ROW row;
int beginyear;
int beginmonth;
int beginday;

if (mysql_query(db, sql)) {
    print_error(mysql_error(db));
    return;
}
if((res = mysql_store_result(db)) == 0) {
    print_error(mysql_error(db));
    return;
}
if ((row = mysql_fetch_row(res)) == 0) {
    print_error("No event found by that number");
    return;
}

sscanf(row[0], "%d-%d-%d", &beginyear, &beginmonth,
        &beginday);
```

Figure 7: Using user-supplied data in MySQL

```
char name[11];
char escapedname[21];

cgiFormStringNoNewlines("name", name, 10);
mysql_real_escape_string(db, escapedname, name, strlen(name));
```

Writing data to the database is more interesting, because of the need to escape the data. Listing 7 shows how it is done.

`escapedname` will hold the same string as `name`, with MySQL special characters escaped so that I can insert them into an SQL statement without worry. It is essential that you escape all strings read from user input. Otherwise a devious person could take advantage of your lapse and do unpleasant things to your database.

7 Debugging CGI Programs

One distinct disadvantage of debugging C is that errors tend to cause a segmentation fault with no diagnostic message about the source of the error. Debuggers are fine for most other types of programs, but CGI programs present a special challenge because of the way they get input.

The `cgic` library includes a CGI program called `capture`. This program saves any CGI input that it sent to it and saves it to a file. You will need to set the file name in `capture`'s source code. When your CGI program needs debugging add a call to `cgiReadEnvironment(char*)` to the top of your `cgiMain()` function. You'll need to set the filename parameter to match the filename set in `capture`.

Send the problematic data to `capture` (either make it the action of the form or the script in your request). You can now use `gdb` or your favorite debugger to see what sort of trouble your code is getting up to.

There are some steps you can take in development to simplify debugging and development. While they apply to all programing, they pay off particularly well in CGI programming.

- A function should do one thing and one thing only.
- Test early and test often.

It's a good idea to test each function you write as soon as possible, to make sure that it performs as expected. It isn't a bad idea to see how it responds to erroneous data as well. It's highly likely that at some point it will be given bad data. Catching this ahead of time can save unpleasant calls on your off hours.

8 Deployment

In most situations your development machine and your deployment machine are not going to be the same. As much as it is possible, try to make your development system match the production system. This isn't always possible. For instance, my software tends to be developed on Linux or OpenBSD, and nearly always deployed on FreeBSD.

When you're preparing to build or install on the deployment machine, it is particularly important to be aware of differences in library versions. You can see which dynamic libraries your code uses with `ldd`. It's a good idea to check, because you will often be surprised what additional dependencies your libraries bring with them.

If library versions are close (e.g. same major number) there probably isn't a big problem. It's not uncommon for deployment and development machines to have incompatible versions if you're deploying to an externally hosted web site.

The solution that I have used is to compile my own, local version of the library. Remove the shared version of the library, and link against this local version rather than the system version. It bulks up your binary, but it removes your dependency on libraries that you don't control.

Once you have built your binary on the deployment system, run `ldd` again to make sure that all of the dynamic libraries have been found. Especially when you are linking against a local copy of a library, it's easy to forget to remove the dynamic version, which won't be found at runtime (or by `ldd`). Keep tweaking the build process, build and recheck until there are no unbound libraries.

9 Speed: CGI vs. PHP

The conventional wisdom has held that a program using the CGI interface will be slower than a program using a language provided by a server module, such as `mod_php` or `mod_perl`. Because I started writing web applications with PHP, I'll use it as my basis for comparison with a CGI program written in C. I make no assertions about the relative speed of C vs. Perl.

The comparison that I used was the external interface to the database (`events.cgi` and `events.php`). Both used the same method for providing interface separation. The internal interface was not tested, since calls to the external interface should dwarf calls to the internal.

Apache Benchmark was used to hit each version with 10,000 queries, just as fast as the server could take it. The C version had a mean transaction time of 581ms, while the PHP version had a mean transaction time of 601ms. With times so close, I suspected that if the tests were repeated, some variation in time would be seen. This proved correct, although the C version was slightly faster more times than not.

My normal development uses a more complex interface separation library, `libtemplate` (<http://www.lazarusid.com/libtemplate.shtml>). I have PHP and C versions of the library. When I compared versions of the event scheduler using `libtemplate`, I found that C had a much more favorable response time. The mean transaction time for the C version was 625ms, not much more than for the simpler version. The PHP version had a mean transaction time of 1,957ms. It was also notable that the

load number while the PHP version was running was generally twice what was seen while the C version was running. There were no users on the system, and no other significant applications running, when this test was done.

The fairly close times of the two C versions tells us that most of the execution time is spent loading the program. Once the program is loaded the program executes quite quickly. PHP, on the other hand, executes relatively slowly. Of course, PHP doesn't escape the problem of having to be loaded into memory. It must also be compiled, a step which the C program has already been through.

10 Conclusions

With the right tools and a little experience, develop with C is no more difficult than perl or PHP. Now that I have the experience and the tools, C is my preferred language for CGI applications.

C excels when the application requires more advanced processing and long term stability. It is not especially susceptible to failure when there are server changes beyond your control, unlike PHP. Short of removing a shared library such as libc or libmysql-client, the C version of our application is hard to break. The speed of execution for C programs makes it a clear choice when the application requires more complex data processing.

11 About the Author

Clay Dowling is the president of Lazarus Internet Development (<http://www.lazarusid.com>). In addition to programming, he enjoys brewing beer and wine. He may be reached via e-mail to clay@lazarusid.com.