# Using C for CGI Programming

**Clay Dowling**

**Abstract**

You can speed up complex Web tasks while retaining the simplicity of CGI. With many useful libraries available, the jump from a scripting language to C isn't as big as you might think.

---

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

I use C in my applications for three reasons: speed, features and stability. Although conventional wisdom says otherwise, my own benchmarks have found that C and PHP are equivalent in speed when the processing to be done is simple. When there is any complexity to the processing, C wins hands-down.

In addition, C provides an excellent feature set. The language itself comes with a bare-bones set of features, but a staggering number of libraries are available for nearly any job for which a computer is used. Perl, of course, is no slouch in this area, and I don't contend that C offers more extensibility, but both can fill nearly any bill.

Furthermore, CGI programs written in C are 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, it does not experience the dramatic changes to which PHP users have been subjected over the past few years.

## The Application

My application is a simple event listing suitable for a business to list upcoming events, say, the meeting schedule for a day or the events at a church. It provides an administrative interface intended to be password-protected and a public interface that lists all upcoming events (but only upcoming events). This application also provides for runtime configuration and interface independence.

I use a database, rather than write my own data store, and a configuration file contains the database connection information. A collection of files is used to provide interface/code separation.

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

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 for any database engine.

### Listing 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 following functions are the minimum necessary to implement the functionality of the administrative interface: list_events(), show_event(), save_event() and delete_event(). I also am 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 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 I can choose a range of events—something I need to do in at least two places.

Next, I need to abstract my records to C structures and 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 here to print all of my source code. Complete source code and my Makefile can be downloaded from my Web site (see the on-line Resources).

## Tools

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

For CGI parsing, I recommend the cgic library from Thomas Boutell (see Resources). 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 also are suitable (see Resources), although I found the Boutell library to be easier to use.

MySQL is pretty much the standard for UNIX Web development, so I 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.

## 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 populates a corresponding array of configuration values, as shown in Listing 2. Now I can populate a string array with whatever keys I've chosen to use and get the results back in the value array.

**Listing 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);

}
```

## User Interface

The user interface has two parts. As a programmer, I'm concerned primarily with the input forms and URL strings. Everybody else cares how the page around my form looks and takes the form itself 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 that are output at the appropriate time. A function that can do this is found in Listing 3.

**Listing 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);

}
```

Before generating output, I need to tell the Web server and the browser what I'm sending; cgiHeaderContentType() accomplishes this task. 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);

## 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 use a couple of functions from the cgic library: cgiFormStringNoNewlines() and cgiFormInteger(). The cgic library implements the main function 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 eventno parameter. cgiFormInteger() retrieves an integer value and sets a default value if no CGI parameter is provided.

I also need to get a whole raft of data from the form in save_event. Dates are thorny things to input because they consist of three pieces of data: year, month and date. I need both a begin and an 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(), a function that 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 use one of the variables that libcgic provides, cgiScriptName. Using this variable instead of a literal one means the program name can be changed without breaking the program.

**Listing 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);
```

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

**Listing 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]();
```

See Resources for information on function pointers. If function pointers still baffle you, 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.

## Database System

MySQL from C is largely the same as PHP, if you're used to that interface. You have to use MySQL's string escape functions to escape problematic characters in your strings, such as quote characters or the back slash character, but otherwise it is basically 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 the MySQL 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() pulls a single MYSQL_ROW variable from 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 want to treat it as numeric data, you 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 use sscanf() to get the components (Listing 6).

**Listing 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);
```

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

**Listing 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));
```

escapedname holds the same string as name, with MySQL special characters escaped so 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.

## 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 acquire input.

To help with this challenge, the cgic library includes a CGI program called capture. This program saves to a file any CGI input sent to it. You need to set this filename in capture's source code. When your CGI program needs debugging, add a call to cgiReadEnvironment(char*) to the top of your cgiMain() function. Be sure to set the filename parameter to match the filename set in capture. Then, send the problematic data to capture, making it either the action of the form or the script in your request. You now can use GDB or your favorite debugger to see what sort of trouble your code has generated.

You can take some steps to simplify later debugging and development. Although these apply to all programming, they pay off particularly well in CGI programming. Remember that a function should do one thing and one thing only, and test early and test often.

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

## Deployment

In most situations, your development machine and your deployment machine are not going to be the same. As much as possible, try to make your development system match the production system. For instance, my software tends to be developed on Linux or OpenBSD and nearly always is 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 this information, because you often may be surprised by what additional dependencies your libraries bring.

If the library versions are close, usually reflected in the 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 I use 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 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 unfound libraries.

## Speed: CGI vs. PHP

Conventional wisdom holds that a program using the CGI interface is 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 use it here 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), because both used the same method for providing interface separation. The internal interface was not tested, as calls to the external interface should dwarf calls to the internal.

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

My normal development uses a more complex interface separation library, libtemplate (see Resources). 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 it was for the simpler version. The PHP version had a mean transaction time of

1,957ms. It also was notable that the load number while the PHP version was running generally was twice what was seen while the C version was running. No users were on the system, and no other significant applications were running when this test was done.

The fairly close times of the two C versions tell 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 also must be compiled, a step that the C program has been through already.

## Conclusions

With the right tools and a little experience, developing CGI applications with C is no more difficult than it is when using 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 server changes are beyond your control, unlike PHP. Short of removing a shared library, such as libc or libmysqlclient, 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.

**Resources for this article:** http://www.linuxjournal.com/article/8058.