CS3157: Advanced Programming

Lecture #4
June 6
Shlomo Hershkop
shlomo@cs.columbia.edu
Overview

Today:
- Introduction to project 1
- Bit of practice
- Beyond basic types
- Some Shell Programming
Project 1

- I will talk about this later….

- Idea: create a cgi program which will let you organize your calendar information

- Project 1: main website and some backend

- Project 2: backend and search engine 😊
Making it easier

- To make it easier we will be coding parts of the project in class together
  - Here to answer you questions
  - Here to help you learn
  - Push you to get it done earlier

- Will still need to work on it on your own etc
- Remember to have readme + Comments!!
Step 1

- Lets start with simple stuff

- Goal: Spitting out the front end…..

- Let me talk about some background stuff and then we can start programming…
Some CGI Environmental Variables

- CONTENT_LENGTH
  - Length of data passed to cgi via post
- CONTENT_TYPE
- QUERY_STRING
- REMOTE_ADDR
  - Ip address of client
- REQUEST_METHOD
- SCRIPT_NAME
- SERVER_PORT
- SERVER_NAME
- SERVER_SOFTWARE
- HTTP_FROM
- HTTP_USER_AGENT
- HTTP_REFERER
- HTTP_ACCEPT
Get vs Post

- When you run CGI as get…url is changed
  - `getenv("QUERY_STRING")`
- Post pushes the information via STDIN
  - `getenv("CONTENT_LENGTH")`
  - `getline(… or equivalent..)`
Printing out quotes

- To print out quotes need " to show up…
Let's get started....

- Let's write a simple CGI program and test it... then adopt it for this step 1....

- Create a C file “project1.c”
int main()
{
    /*cgi so lets print out */

    printf( "Content-type: text/html\n\n" );

    /* lets check if we are processing inputs */

    printf("<html><head><title>Test 1</title></head> <body>");

    if( getenv("QUERY_STRING") !=NULL )
    {
        printf("Am processing input");
    }
    else{
        printf("Welcome first time<P>");
    }
    printf("</html>");
} /* end main */
next

- gcc -o project1.cgi.exe project1.c
- Remember it has to be in cgi-bin

- Start abyss

- http://localhost/cgi-bin/project1.cgi.exe
It should show welcome

Now load:

http://localhost/cgi-bin/project1.cgi.exe?ss

Should see other message…
next

- Lets replace the welcome with a function call:

- Add this to your file:
  - `void showFirstForm();`
if( getenv("QUERY_STRING") !=NULL )
{
    printf("Am processing input");
}
else{
    showFirstForm();
}
Code should look like:

```c
void showFirstForm(){

    printf("<form action="project1.cgi.exe" method="GET"> <p>");
    printf(" Please enter some text: <input type="text" name="string"></p>" );
    printf("<input type="submit">" );
    printf("</form>" );

}
```
- Compile the code
- Enter some input and press the button
- What happens ??
Now change the form to show user password login

Remember that the password shouldn’t show up … instead of text change type to “password” (remember to escape quotes)
next

- Replace the “am processing input with call to processLogin();

- For today, want to simply print out the username and password received

- Need to scan through the string and split according to & and =

- Create variables as needed

- Use malloc to allocate space once you count…but don’t forget +1 for slash zero at end
Code please

- **man getenv**
  - Will show you what is returned

- `char *qstring = getenv("QUERY_STRING")`

- Remember all variables have to be created at top
At this point should have code and program which when loaded shows a login form, when inputted, shows the input received for GET
Next change

- Allow the user to enter a password and display it and the user name

- Code example
Add functionality for going through a password file and adding the user and md5 of password to it so that we can later check if the user/password matches…
Next step

- Copy code to project1b.c

- (on your free time) Adopt for POST
  - Remember will be reading STDIN
  - NEED TO CHECK INPUT_LENGTH
Back to c...
Creating your own types

- Equivalent to a class idea in other programming languages, you can define your own types in C

```c
struct name {
    types
}
```
example

```c
struct point {
    int x;
    int y;
}

Usage:
struct point a;
a.x = 5;
a.y = 10;
```
Anonymous structs

- Can also create anonymous structs

```c
struct {
    int x;
    int y;
} a, b;
```
Nesting

```c
struct rect {
    struct point pt1;
    struct point p2;
}
```

- Use:
  ```c
  struct rect largeScreen;
  ```
Making space

- Remember in the proceeding examples, simple types so memory is automatically allocated (in a sense).

- struct student {
  char * name;
  int age;
}

  struct student a;
  a.name = (char*)malloc(sizeof(char)*25));
  ...

Use in functions

```c
struct point makePoint(int x, int y)
{
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```
Operations

- Copy
- Assignments
- & (addressing)
- Accessing members

- How do we compare 2 structs
Structs and pointers

- `struct point *example = (struct point *)malloc(sizeof(struct point));`

- `(*example).x`

  what does
  `*example.x` mean?

  Shortcut:
  `example->x`
typedef

- defining your own types using typedef (for ease of use)

```c
typedef short int smallNumber;
typedef unsigned char byte;
tyedef char String[100];

smallNumber x;
byte b;
String name;
```
enum

- define new integer-like types as enumerated types:
  ```c
  enum weather { rain, snow=2, sun=4 };  
  typedef enum {Red, Orange, Yellow, Green, Blue, Violet} Color;
  ```

- look like C identifiers (names)
- are listed (enumerated) in definition
- treated like integers
  - start with 0 (unless you set value)
  - can add, subtract — e.g., color + weather
  - cannot print as symbol automatically (you have to write code to do the translation)
enum

- just fancy syntax for an ordered collection of integer constants:
  
  ```c
  typedef enum {
  Red, Orange, Yellow
  } Color;
  ```

- is like
  
  ```c
  #define Red 0
  #define Orange 1
  #define Yellow 2
  ```

- here’s another way to define your own boolean:
  
  ```c
  typedef enum {False, True} boolean;
  ```
Usage

enum Boolean {False, True};

... enum Boolean shouldWait = True;
...
if(shouldWait == False) { .. }
struct

int main() {
struct {  
    int x;
    char y;
    float z;
} rec;
rec.x = 3;
rec.y = 'a';
rec.z = 3.1415;
printf( "rec = %d %c %f\n", rec.x, rec.y, rec.z );
} // end of main()
struct

int main() {
struct record {
int x;
char y;
float z;
};
struct record rec;
rec.x = 3;
rec.y = 'a';
rec.z = 3.1415;
printf( "rec = %d %c %f\n", rec.x, rec.y, rec.z );
} // end of main()
int main() {
typedef struct {
int x;
char y;
float z;
} RECORD;

RECORD rec;
rec.x = 3;
rec.y = 'a';
rec.z = 3.1415;
printf( "rec = %d %c %f\n", rec.x, rec.y, rec.z );
} // end of main()
note the use of malloc where “sizeof” takes the struct type as its argument (not the pointer!)

```c
int main() {
    typedef struct {
        int x;
        char y;
        float z;
    } RECORD;
    RECORD *rec = (RECORD *)malloc( sizeof( RECORD ));
    rec->x = 3;
    rec->y = 'a';
    rec->z = 3.1415;
    printf( "rec = %d %c %f\n", rec->x, rec->y, rec->z );
} // end of main()
```
Important to understand

- overall size of struct is the sum of the elements, plus padding for alignment (i.e., how many bytes are allocated)
- given previous examples: sizeof( rec ) -> 12
- but, it depends on the size and order of content (e.g., ints need to be aligned on word boundaries, since size of char is 1 and size of int is 4):

```c
struct {
    char x;
    int y;
    char z;
} s1;
/* x y z */
/* |----|----|----| */
/* sizeof s1 -> 12 */

struct {
    char x, y;
    int z;
} s2;
/* xy z */
/* |----|----| */
/* sizeof s2 -> 8 */
```
Reminder

- Pointers to structs are common — especially useful with functions (as arguments to functions or as function type)
- Two notations for accessing elements: (*sp).field or sp->field
- (Note: *sp.field doesn’t work)

```c
struct xyz {
  int x, y, z;
};
struct xyz s;
struct xyz *sp;
...
s.x = 1;
s.y = 2;
s.z = 3;
sp = &s;
(*sp).z = sp->x + sp->y;
```
Arrays of structs

- notations for accessing elements: arr[i].field

```c
struct xyz {
    int x, y, z;
};
struct xyz arr[2];
...
arr[0].x = 1;
arr[0].y = 2;
arr[0].z = 3;
arr[1].x = 4;
arr[1].y = 5;
arr[1].z = 6;
```
unions

- union
- like struct:
  union u_tag {
  int ival;
  float fval;
  char *sval;
  } u;

- but only one of ival, fval and sval can be used in an instance of u (think container)
- overall size is largest of elements
Example

#define NAME_LEN 40

struct person {
    char name[NAME_LEN+1];
    float height;
};

int main( void ) {
    struct person p;
    strcpy( p.name,"suzanne" );
    p.height = 60;
    printf( "name   = [%s]\n", p.name );
    printf( "height = %5.2f inches\n", p.height );
} // end of main()