Overview of Web Applications and C

Alexander Berenbeim

2016-09-20 Tue

Outline

Mobile App/ Web App/ Desktop

- How do we intend on users to interact with this doodad?

Ideally we would have an application on mobile devices which:

- sends push notifications to the specific graduate students who can possibly fill a sub request
- can be updated once one TA accepts the sub request.
- keeps track of who has hours owed

Mobile app development requires platform commitments

- For now, let's focus on a desktop and web interface.

User Interface

User Login & User Information Submission

- Structuring user data

Controlling Requests

- Time and Course Request
- Sending Sub-request for targetted TAs with hours saved/owed by TA making request
- Accepting/Declining Subrequest
- Cancelling Requests
- Confirming Requests (Indicating that someone has fulfilled the request)

Backend Processes

- Gather/Update work status of Graduate Students
- eg some students may not be able to work as graduate students for a term, but may be able to later
- Gather/Update Current TA Schedule
- Gather/Solicit/Update Current Course Schedule of TAs
- Gather/Solicit/Update Current Seminar Schedule of TAs
- Maintaining log of hours owed/ saved
- Transferring hours worked/owed
- Contacting only those who meet the request criterion

What Distinguishes A Website from a Web App

- Doesn't really exist except for static html demarcating a "web site"
- web apps have functionality similar to software or mobile app (What does this even mean in principle?)
- HTML5 and App Embedding
- Most web applications are 3-tiered

3-tiered web applications

- A user with a web browser provides inputs to be processed by an application logic that interacts with a database, often computing outputs for the user and side effects updating the data base

The three tiers here are Presentation, Application, Storage

- Presentation : Browser
- Application : application logic, engine using dynamic web content
- Storage : Database

What Advantages Are There To Using C Over All Other Languages For This Project?

- None. Absolutely none
- In fact, C may be one of the least efficient and advisable languages to pursue app development in for a mobile or web environment because of the difficulty in writing good code, even for expert programmers. Even more than C++, which has zero-overhead for abstractions and has direct mapping of hardware features from C
- C is good for low level work where data integrity matters
- The assembler is portable and C is good for tiny executables
- Writing Good C Code Requires Discipline
- Learning Good Programming Practices Cannot Occur In A Vacuum

The Components of a C Program

Preprocessor Commands

- ex: "#include <stdio.h>" tells the C compiler to include stdio.h before compiling

Functions

- ex: "printf(...)" is a function in C which displays the string inside

Variables

Statements & Expressions

Comments

- "/ * The * should be next to the / * /"

Some Key Data Types

Void type

Arithmetical types

Enumerated Types

Derived Types

- Pointers
- Arrays
- Structure
- Union
- Function

Void Type

Function returns as void

- if you want to run a function for a side-effect and not have a value returned

Function arguments as void

- used when a function has no parameters, e.g. "int rand(void)"

Pointers to void

- used to represent the address of an object but not its type

Pointers

- Physically are a group of cells holding address information
- Careless use of pointers leads to unintelligible C code
- One hallmark of good C code is a conceptually clean use of pointers
- Pointers are often the only way to express a computation
- Pointers are faster to use than arrays
- The general form a pointer variable declaration "type *var-name;"
- We use the unary * operator to return the value of a variable located at the address specified by the operand

Uses of pointers

- Pointer arithmetic
- Array of pointers
- Pointer to Pointer
- Passing pointers to functions
- Returning pointers from functions: in C functions can return a pointer to the local variable, static variable, and dynamically allocated memory.

Example: swap(a,b)

- C passes arguments to functions by value, so we can't affect the arguments called by a routine
- Pointer arguments enable a function to access and change objects in the function calling the argument

Bad

```
void swap (int x, int y)
{
int temp;
temp = x;
x=y;
y = temp;
}
```

Good

```
void swap(int *px,int *py)
{
int temp;
temp = *px;
*px = *py ;
*py = temp ;
}
```

- The good program must swap copies of our arguments.

Example: Pointers Arrays

- Suppose we have a private array of character strings, like the names of months and we want to write a function that given an integer will return the appropriate month. Consider:
- ** month-name : return name of the n-th month **

```
char *month _ name(int n) { static char *name[] =
{
"Illegal month", "January", "February",
"March", "April", "May", "June", "July",
"August", "September", "October", "November",
"December"
};
return (n < 1 || n >1) ? name [ 0 ] : name [ n ];
}
```

Arrays versus pointer arrays

- In C all multidimensional arrays are really one-dimensional arrays whose elements are arrays
- multidimensional arrays are slower than pointer arrays
- multidimensional arrays are all of fixed length whereas variable arrays may be of different length

"int a[10][10]" versus "int *b[10]"

- both a[2][4] and b[2][4] are valid, but a has 100 int sized locations (with a subscript calculation 10r+c used to find an element a[r][c]), while b stores 10 pointers of arbitrary length
- Because pointers can be of arbitrary length, care **must** be taken to consciously manage resource constraints

Structures

- Arrays can be used to define a type of variables with different items of the same kind
- Structures can be used to combine data items of different kinds
- Structures can be passed as a function argument like variables or pointers
- We'll build our program around pointers and structures

Structure format

struct [structure tag]{
member definition;
member definition;
...
member definition;
} [one or more structure variables];

Example:

```
struct Paper {
char title[ 100 ];
char author[ 100 ];
char subject[ 100 ];
} thesis;
```

The structure tag for Paper is optional

We access members of a structure with the member access operator (.)

- "thesis.title"

Example

```
struct TA {
char name[ 100 ];
char *courses[ 6 ];
char *seminars[ 6 ];
char *classes[ 6 ];
char email[ 100 ];
};
```

Pointers to structure

- We can define a pointer to a structure the same way we define a pointer to other variables "struct Paper *struct _ pointer;"
- We can find the address of a structure variable using the unary '&' operator, "struct _ pointer = &paper1;"
- We can access the members of a structure using a pointer to that structure with the "->" operator, "struct _ pointer->title"

Example: Pointers to Structure

```
#include <stdio.h>
#include <string.h>
/ * function declaration * /
void printPaper (struct Paper *paper );
int main ( ) {
struct Paper paper1; / * Declare paper1 of type Paper * /
struct Paper paper2; / * Declare paper2 of type Paper * /
/ * paper1 specification * /
strcpy ( paper1.title, "Web Apps");
strcpy ( paper1.author, "Alex");
strcpy ( paper1.subject, "Getting a Job");
/ * paper2 specification * /
strcpy ( paper2.title, "Basics of C");
strcpy ( paper2.author, "Berenbeim");
strcpy ( paper2.subject, "Doing a Job");
/ * print paper1 info by passing address of paper1 */
printPaper( &paper1 );
/ * print paper2 info by passing address of paper2 */
printPaper( &paper2 );
return 0;
}
```

Example: Pointers to Structure (continued)

```
void printPaper( struct Paper *paper ) {
printf( "Paper title : %s", paper -> title);
printf( "Paper author : %s", paper -> author);
printf( "Paper subject : %s", paper -> subject);
}
```

Once compiled and executed, this code produces:

```
Paper title : Web Apps
Paper author : Alex
Paper subject : Getting a Job
Paper title : Basics of C
Paper author : Berenbeim
Paper subject : Doing a Job
```