CS3157: Advanced Programming

Lecture #10
Mar 20
Shlomo Hershkop
shlomo@cs.columbia.edu

Announcement

• Welcome back from spring break
• Hope you’ve caught up with your courses 😊
• Have the exams back, will return at the end of class
Announcements

• Based on feedback for the lab component:
  – Reading
  – Comprehension
  – Will try to include relevant reading for lab, but the easiest way to get started
    • Look over the class notes (BEFORE)
    • Read the instructions
    • Ask me ← ← ←

Outline

• Much more C
  – Pointers
  – Const
  – Typedef
  – Union
  – Enum

• Reading:
  – K & R 5.5-,6
  – Deitel chapter 7
Outline for rest of semester

- Will cover basic C then basic/advanced C++
- Shell programming
  - Useful
  - Looks like it's coming to Windows soon (finally)
- PHP / webscripting
- Advanced topics

Pointers

- Make sure you feel comfortable with the idea of what is happening inside pointer

- Will try to use examples today to make specific points
int main(){
    int number = 10;
    foo(&number);
    return 0;
}

void foo(int *p){
    *p = 30;
}

Question

• What's the advantage of passing in by pointer reference?

• What is the problem?

• How would we solve it?
const

- Allows the compiler to know which values shouldn’t be modified
- Added in to C later

- Example:
  ```
  const int a = 5;
  void foo(const int x) {
  }
  ```

const

- Better than #define since error message will be easier to understand since preprocessor not involved
- Very useful in functions to either return const or make sure a pointer doesn’t alter the original object
Const pointer to non-const

- This is a pointer which always points to the same location, but the value can be modified

- int * const ptr = &x;
  
  *ptr = ??
  can’t say
  ptr = & ??

- Example2: array name

Const pointer to const data

- Int x = 200;
- const int * const ptr = &x;
• Some confusion
  – int const * X
  – const int * X //variable pointer to const
  – int * const Y //const pointer to int
  – int const * const Z //const point to const

Pointers to functions

• C allows you to also pass around a pointer to a function
  • void foo (int , int (*) (int , int) );
  • int example1(int x, int y) { return x+y; }
  • foo(5, example1);
void foo(int a, int (*A)(int,int)){
    if((*A)(5,10) > 0){
    } else {
    }
}

Creating your own types

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

struct name {
    types
}
example

struct point {
    int x;
    int y;
}

• Usage:
    struct point a;
    a.x = 5;
    a.y = 10;

Anonymous structs

• Can also create anonymous structs

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).
- ```c
  struct student {
      char * name;
      int age;
  }
  ```

  ```c
  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)

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

  smallNumber x;
  byte b;
  String name;
enum

• define new integer-like types as enumerated types:
  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:
  typedef enum {
    Red, Orange, Yellow
  } Color;
• is like
  #define Red 0
  #define Orange 1
  #define Yellow 2

• here’s another way to define your own boolean:
  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()
```c
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
", 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 */ /* 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:

```c
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

```c
#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()
```
For Next Class

• Do relevant reading
• Look over your exam, please see me if you don’t understand/have questions
  – See you in lab Wednesday

Over view of assignment

• Extend the lab example
• Integrate perl in c and cgi
• Work with graphics
• Have something cool to show off to your friends or on interviews.

• Hints: if you are sending too much time….ask for help
  – examples