CS 261 – Data Structures

Introduction to C Programming

Notes by
Tim Budd
Ronald Metoyer
Sinisa Todorovic
Why C?

• C is a simple language,
• C makes it easier to focus on important concepts
• Java is a high level, object oriented language,
• C is a lower level, imperative language
• Important to learn both
• Lays groundwork for many other languages
Comparing Java and C

• C does not have:
  – Classes
  – Inheritance
  – Polymorphism
  – Function overloading
  – Garbage collection
  – Array bounds checking
Comparing Java and C

• Functions in C are the equivalent of functions in classes
• Arrays and structures in C are the major data storage mechanisms
• Pointers!! Lots of pointers.
Separation of Interface and Implementation

• Interface files (\* . h) have
  – declarations, preprocessor commands
  – function prototypes -- header but no body:
    • Example: int max(int a, int b);
    terminated with a semicolon

• Implementation files (\* . c) have implementations
Preprocessor

• A preprocessor scans C programs before compiling

• Used to:
  – Include/embed other files (usually used to embed .h files)
  – Make symbolic constants
  – Conditionally compiles code
  – Other uses
#define MAX 423  /* Replaces MAX in code with 423. */

#if (MAX < 3)
  .
  . /* Conditional code here compiles only when #if evaluates to true */
#endif
Ensuring Declarations Seen Only Once

/* Interface file for foo.h */

#ifndef BIG_COMPLEX_NAME
#define BIG_COMPLEX_NAME
.
.
/* Rest of foo.h file. */
.
#endif

If foo.h is included more than once (e.g., through other included files), it only gets evaluated once
Variables and Declarations

• When you declare a variable, a memory space is reserved for that variable

```c
int i;    /* 8 bytes for 64-bit machine */
double   d;
long test[100];/* reserved 100 locations of size long */
test[0] = 2;    /* OK command */
test[99] = 3;   /* OK command */
test[100] = 4;  /* ERROR !!!*/
```
What is elType?

• elType is a symbolic constant, and represents the type of value

    # define elType double
How Big is Struct?

• As large as the combined data fields.

```c
printf("size is %d\n", sizeof(long));
```

`printf` takes a formatting string and list of values:

- `%d` → integers
- `%f` `%g` → floats and doubles
- `%c` → characters
- `%s` → strings
- `%%` → percent sign
How Big is Struct?

• As large as the combined data fields.

\[ \text{printf("size is \%d\n", sizeof(long));} \]

pseudo-function that takes as argument \textit{TYPE}, and tells you how many bytes it needs
Function Definitions

Functions look like those in Java, but are not a part of a class:

```c
returnType functionName(arguments) {  
    variable-declarations; /*!<Must come first*/
    function-body;
}
```
Function Definitions

Example: return sum of integer elements:

```c
long arrSum(int arr[], unsigned int n)
{
    unsigned int i; /* Loop variable. */
    long sum = 0;  /* Sum initialized to zero. */
    for (i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Need to pass size of array (not included in `arr`).
Two Level Scope

There are two levels of scope:

– **Global**: variables declared outside of any function
  
  • Use sparingly

– **Local**: variables declared inside of a function.
  
  • Local variable declarations must be listed first, before statements.
Two Level Scope -- Example

double avg;
/* Global variable: can access in any function. */

void arrAvg(int arr[], unsigned int n)
{
    unsigned int i;
    /* Local variables: access only within function */

    long sum = 0;
    for (i = 0; i < n; i++)
        sum += arr[i];
    avg = (double)sum / n;  /*Casting*/
}

Variable and its Memory Location

```c
double mass; /* variable */
long memory; /* variable */

mass = 0.01;

memory = & mass;

printf("%e, %p \n", mass, memory);

Output: 1e-2, ffbff958
```
Parameters: **Pass-By-Value Only**

- Arguments to functions are passed by value

- Parameters are initialized with a COPY of the argument

- Simulate pass-by-reference using pointers
Two Level Scope -- Example

double test;
/* Global variable: can be accessed in any function. */

void printing(void){
    int n=5;
    assignment(n);
    printf("n=\%d, test=\%d",n,test);
}

void assignment(int n){
    n++;
    test++;
}
Pointers

• Pointers in C are explicit (implicit in Java)

• A pointer is a variable that refers to a memory location
Pointer Value vs. Thing Pointed At

the value of the pointer

vs.

the value of the thing the pointer points to:

\[
pVal \rightarrow D3C2 \rightarrow Value \text{ at location } D3C2 \rightarrow 42 \leftarrow \star pVal
\]
Pointers

```c
int *pVal;  /* Pointer (uninitialized) to unallocated integer value. */
```
**Pointers**

```c
int *pVal; /* Pointer (uninitialized) to unallocated integer value. */

pVal = 0; /* Initialize pointer to indicate that it is not allocated. */
```

![Diagram of a pointer indicating a "null" pointer.](image-url)
Pointers

```c
int *pVal; /* Pointer (uninitialized) to unallocated integer value. */

pVal = 0; /* Initialize pointer to indicate that it is not allocated. */

/* Allocate unassigned integer and */
/* assign memory address to pVal. */
pVal = (int *) malloc(sizeof(int));
```
Points

```c
int *pVal; /* Pointer (uninitialized) to unallocated integer value. */

pVal = 0; /* Initialize pointer to indicate that it is not allocated. */

/* Allocate unassigned integer and assign memory address to pVal. */
pVal = (int *) malloc(sizeof(int));
*pVal = 42;
```
Pointer Value vs. Thing Pointed At

the value of the pointer

vs.

the value of the thing the pointer points to

pVal

* pVal

42

Value

Pointer
Pointer Syntax

• Use * to
  – declare a pointer,
  – get value of pointer

• Use & to get address of a variable

  double *ptr;
  double pi, e;
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
ptr = &e;
*ptr = 2.71828;

printf("Values: %p %g %g %g\n", ptr, *ptr, pi, e);
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;

ptr = &e;
*ptr = 2.71828;

printf("Values: %p %g %g %g\n", ptr, *ptr, pi, e);
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
ptr = &e;
*ptr = 2.71828;

printf("Values: %p %g %g %g\n", ptr, *ptr, pi, e);
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
ptr = &e;
*ptr = 2.71828;

printf("Values: %p %g %g %g\n", ptr, *ptr, pi, e);
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
ptr = &e;
*ptr = 2.71828;

printf("%p %g %g %g\n",
ptr, *ptr, pi, e);
double *ptr;
double pi, e;

ptr = &pi;
*ptr = 3.14159;
ptr = &e;
*ptr = 2.71828;

printf("%p %g %g %g\n",
      ptr,   *ptr,   pi,    e);

Output: ffbff958 2.71828 3.14159 2.71828
Pointer Syntax

- Use \[\] to declare a pointer
- Use \[0\] to get the value of pointer

```c
double data[]; /*pointer*/
double value = 5.3; /*variable*/
data = & value;
printf("%g", data[0]);
```
Structures

Structures are like classes that have only public data fields and no methods:

```c
struct Gate {
    int type;    /* Type of gate. */
    struct Gate *left; /* Left input. */
    struct Gate *right; /* Right input. */
};
```
Accessing Data Fields

Access to data fields uses the same dot notation you are used to:

```c
struct Gate gate;

gate.type = 3;
```

but often combined with pointers …
Pointers and Structures

Pointers often point to structures.

```c
struct Gate *p;
struct Gate g;

p = &g;
p->type = 3; /* Set Gate type of struct that p points to. */
```
Pointers and Structures

Pointers often point to structures.

```c
struct Gate *p;
struct Gate g;

p = &g;
p->type = 3; /* Set Gate type of struct
               that p points to. */
/* Same as  (*p) . type = 3        */
/* Same as   g . type = 3         */
```
Pointers and Pass-by-Reference Parameters

Passing a reference parameter using a pointer

```c
void set_pi(double *p) {
    *p = 3.14159;
}
```
Pointers and Pass-by-Reference Parameters

Passing a reference parameter using a pointer

```c
void set_pi(double *p) {
    *p = 3.14159;
}

double d = 2.718281;
set_pi(&d); /* Get pointer to d and pass it to set_pi */
printf("d = %g\n", d);
```
Structures and Pass-by-Reference Parameters

Very common idiom:

```c
struct Vector vec; /* Note: not pointer */

vectorInit(&vec); /* Pass by reference */

vectorAdd (&vec, 3.14159);
```
Arrays Always Passed by Reference

```c
void foo(double d[]) { /* Same as
   foo(double *d) */
   d[0] = 3.14159;
}
.
.
double data[4];
data[0] = 42.0;
foo(data); /* Note: NO ampersand. */
printf("%g",
data[0]);
```
Dynamic Memory

- No `new` operator
- Use `malloc(num-of-bytes)` instead
- Use `sizeof` to figure out how big (how many bytes) something is

```c
struct Gate *p = (struct Gate *) malloc (sizeof(struct Gate));
assert(p != 0); /* Always a good idea. */
```
Check Conditions: `assert`  

- We will use `assert` to check all sorts of conditions  
- Halts program if condition not found

```c
#include <assert.h>

/* Assert checks if specified condition is true. */
assert(whatever-condition);
```
Side Note: No Boolean Type

- Standard C (C89) does not have a boolean data type
- Can use ordinary integer: test is zero (false) or not zero (true)
- Can also use pointers: test is null/zero (false) or not null (true)

```c
int i;
if (i != 0) ...
if (i) ... /* Same thing. */

double *p;
if (p != 0) ...
if (p) ... /* Same thing. */
```
Questions ??
Next Lecture

• Read Chapter 5 on ADTs
• Do your own review of Big-OH and Algorithms (See posted reading and worksheets on the schedule)