CS 261 – Data Structures

Introduction to
C Programming
Why C?

• C is a simple language,
• C makes it easier to focus on important concepts
• C is a lower level, imperative language
• Lays groundwork for many other languages
Files: Interface and Implementation

- Interface files (*.h) have
  - Declarations of variables
  - Declarations of types,
  - Preprocessor commands,
  - Function prototypes -- header but no body:
    - Example: `int max(int a, int b);`
      terminated with a semicolon!

- Implementation files (*.c) have implementations
• 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 !!!*/
```
Declarations of Types

• For example, a symbolic constant `elType’ is declared as type double

    # define elType double
Preprocessor

- A preprocessor scans C programs before compiling

- Used for:
  - Including other .h files
  - Defining a symbolic constant
  - Conditionally compiling a code
  - Other
Preprocessor -- Examples

#include <stdio.h>
/* Including other .h files */

#define MAX 423
/* Replaces MAX in code with 423. */

#if (MAX < 3)
/* Conditional code here that will be compiled only when #if evaluates to true */
#endif
If `foo.h` is included more than once (e.g., through other included files), it only gets evaluated once
Function Definitions

returnType functionName(arguments) {
    variable-declarations; /*Must come first*/
    function-body;
}

Function Definitions -- Example

Return a sum of n integers:

```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`).
Scope of Variables

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.
Scope of Variables -- Example

double avg;
/* Global variable: can be accessed 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*/
}

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

mass = 0.01;

memory = & mass;

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

Output: 1e-2,  ffbff958
Function Arguments: Pass-By-Value

• Arguments to functions can be passed by value

• A copy is formed in the function, initialized to the argument value
Pass-By-Value -- 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){ /* pass n by value */
    n++;
    test = n;
}

Output: ?
Function Arguments: Pass-by-Reference

Passing a reference using a pointer

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

double d = 2.718281;
set_pi(&d); /* Pass \texttt{d} by reference */
printf("d = \%g\n", d);

Output: ?
void set_pi(double *p) {
    *p = 3.14159;
}

double d = 2.718281;
set_pi(d); /* Pass d by value */
printf("d = %g\n", d);

Output: ?
Pointers

• Pointers in C are explicit

• 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:

```
D3C2
```

Value at location D3C2

```
pVal
D3C2
```

```
*pVal
42
```

```
Pointer
```
Pointers

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

int *pVal; /* Pointer uninitialized to unallocated integer value. */

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

Pointer

pVal

Indicates a “null” pointer.
Pointers

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

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

```c
/* Allocate integer and */
/* assign memory address to pVal. */
```

```c
pVal = (int *) malloc(sizeof(int));
```
Pointers

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

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

/* Allocate 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
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);

Output: ?
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
Alternative 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]);
```

Output: ?
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 in the Structure

```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 g.type 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 g.type that p points to */

/* Same as  (*p).type = 3 */
/* Same as  g.type = 3 */
```
Very common idiom:

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

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

vectorAdd (&vec, 3.14159);
```
void 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

• Use `malloc(num-of-bytes)`
• Use `sizeof` to figure out 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. */
```
Next Lecture

• Read Chapter 5 on ADTs
• Big-OH and Algorithms
• See posted reading and worksheets