C Parameter Passing Notes

- We'll say formal parameter vs actual parameter.
- Formal parameters are placeholders in function definition.
- Actual parameters (aka arguments) actually have a value.
- In C, all parameters are passed by value.
- Parameter passing by reference is simulated by passing the address of the variable.
  ```c
  scanf("%d", &n);
  ```
- Array names represent the address of the array. In effect, arrays are passed by reference.
  ```c
  int UpdateArray (int A[], int n) {
      A[0] += 5;
      ...
  }
  ```

Adapted from Dennis Frey, CMSC 313 Spring 2011

Characters, Strings & Pointers
Strings revisited

Recall that a string is represented as an array of characters terminated with a null (\0) character.
As we’ve seen, arrays and pointers are closely related. A string constant may be declared as either:

- char[] or char *

  ```c
  char hello[] = "Hello Bobby";
  ```

  or (almost) equivalently:

  ```c
  char *hi = "Hello Bob";
  ```

A typedef could also be used to simplify coding:

```c
typedef char* STRING;
STRING hi = "Hello Bob";
```

Arrays of Pointers

Since a pointer is a variable type, we can create an array of pointers just like we can create any array of any other type.

Although the pointers may point to any type, the most common use of an array of pointers is an array of char* to create an array of strings.

Boy’s Names

A common use of an array of pointers is to create an array of strings. The declaration below creates an initialized array of strings (char *) for some boys names. The diagram below illustrates the memory configuration.

```c
char *name[] = { "Bobby", "Jim", "Harold" };
```

<table>
<thead>
<tr>
<th>name:</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td>o</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>J</td>
<td>i</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
<td>a</td>
<td>r</td>
</tr>
</tbody>
</table>
Command Line Arguments

Command line arguments:

```
/a.out breakfast lunch dinner
```

These arguments are passed to your program as parameters to `main`.

```c
int main(int argc, char argv[])
```

`argc` is the number of command line arguments

`argv` is an array of `argv` strings.

`argv[0]` is always the name of your executable program.

The rest of `argv[1]` are the remaining strings on the command line.
**Command Line Arguments (2)**

Example, with this command at the Linux prompt:

```bash
myprog hello world 42
```

we get:

```c
argc = 4
argv[0] = "myprog"
argv[1] = "hello"
argv[2] = "world"
argv[3] = "42"
```

Note: `argv[3]` is a string NOT an integer. Convert using `atoi()`:

```c
int answer = atoi(argv[3]);
```

** structs & Pointers **

```c
int *name;
/* pointer declaration */
int age = 42;

*name += 12;
printf("My age is %d\n", *name);
```

** Reminder **

You can't use a pointer until it points to something. Just declaring a variable to be a pointer is not enough:

```c
int *name; /* pointer declaration */
int age = 42;

*name += 12;
printf("My age is %d\n", *name);
```
Pointers to Pointers

A pointer may point to another pointer.

Consider the following declarations:

```c
int age = 42; /* an int */
int *pAge = &age; /* a pointer to an int */
int **ppAge = &pAge; /* pointer to pointer to int */
```

Draw a memory picture of these variable and their relationships.

What type and what value do each of the following represent?

```c
age, pAge, ppAge, *pAge, **ppAge
```

---

pointers2pointer.c

```c
int main ()
{
    /* a double, a pointer to double, */
    /* and a pointer to a pointer to a double */
    double gpa = 3.25, *pGpa, **ppGpa;
    /* make pGpa point to the gpa */
    pGpa = &gpa;
    /* make ppGpa point to pGpa (which points to gpa) */
    ppGpa = &pGpa;
    // what is the output from this printf statement?
    printf("%.2f, %.2f, %.2f", gpa, *pGpa, **ppGpa);
    return 0;
}
```

---

Pointers to struct

```c
typed struct student
{
    char name[50];
    char major[20];
    double gpa;
};

STUDENT bob = {"Bob Smith", "Math", 3.77};
STUDENT sally = {"Sally", "CSEE", 4.0};
STUDENT *pStudent;
    /* pStudent is a "pointer to struct student" */
pStudent = &bob; /* make pStudent point to bob */
    // use printf to access the members */
printf("%s\n", pStudent->name);
printf("%.2f\n", pStudent->gpa);
    /* make pStudent point to sally */
pStudent = &sally;
printf("%s\n", pStudent->name);
printf("%.2f\n", pStudent->gpa);
}```
### Pointer in a struct

The data member of a struct can be a pointer:

```c
#define FNSIZE 50
#define LNSIZE 40
typedef struct name {
    char first [FNSIZE + 1 ];
    char last [ LNSIZE + 1 ];
} NAME;
typedef struct person {
    NAME *pName; // pointer to NAME
    int age;
    double gpa;
} PERSON;
```

### Pointer in a struct (2)

Given the declarations below, how do we access bob’s name, last name, and first name?

Draw a picture of memory represented by these declarations:

```c
NAME bobsName = { "Bob", "Smith"};
PERSON bob;
bob.age = 42;
bob.gpa = 3.4;
bob.pName = &bobsName;
```

### Self-referencing structs

Powerful data structures can be created when a data member of a struct is a pointer to a struct of the same kind. The simple example on the next slide illustrates the technique.
typedef struct player {
    char name[20];
    struct player *teammate; /* can't use TEAMMATE yet */
} TEAMMATE;

TEAMMATE *team, bob, harry, john;
team = &bob; /* first player */
strcpy(bob.name, "bob", 20);
bob.teammate = &harry; /* next teammate */
strcpy(harry.name, "harry", 20);
harry.teammate = &john; /* next teammate */
strcpy(john.name, "bill", 20);
john.teammate = NULL; /* last teammate */

/* typical code to print a (linked) list */
/* follow the teammate pointers until ** NULL is encountered */
// start with first player
TEAMMATE *t = team;
// while there are more players...
while (t != NULL) {
    printf("%s\n", t->name);
    // next player
    t = t->teammate;
}
**CONST POINTERS**

4 ways to declare pointers in combination with const:

int *ptr; // no restriction
const int *ptr; // can't change *ptr
int * const ptr; // can't change ptr
const int * const ptr; // can't change either

Mostly used with formal parameters.