Recap...

☐ File I/O
☐ Function prototypes
☐ Debugging
☐ Variable scope
Today

- More debugging
- More pointers

Preprocessors

- I already went over these two but here is a recap, and some more detail
- `#include`
  - `/usr/include` – `stdio.h`, `stdlib.h`, `math.h`, `string.h`, `ctype.h`, `limits.h`
  - If you use `include` `math.h`, then you need a `-lm` at the end of your compile command
Preprocessors II

- #define
  - convention – in caps
  - You can define **macros** as well
  - #define FOO bar
  
  - #define FOR_ALL for (i=0; i<ARRAY_SIZE; i++)
    
    ... FOR_ALL {
      data[i] = 0;
    }

- #define SQR(x) ((x)*(x))
  - note the extra parentheses

- Both define and include end at EOL, however, you can continue with a \n

Preprocessors III

- #ifdef (pg. 146) + ifndef, undef, endif, else
  - Conditional compilation
  
  #ifdef DEBUG
  
  printf ("The value of the variable: %d\n",
          val);

  #endif

- Now you can use #define DEBUG or undef DEBUG
Preprocessors IV

- **#ifndef**
  - Allows for code to be compiled if symbol is *not* defined.
  ```c
  #ifndef DEBUG
  printf("This is production code");
  #endif
  ```

- **#else**
  ```c
  ifdef DEBUG
  printf("This is test code");
  #else DEBUG
  printf("This is production code");
  endif
  ```

- You can use these techniques to debug as well as write regular code
  - Helps in commenting
  - /* lots of code */

Preprocessor V

- You can use these techniques to debug as well as write regular code
  - Helps in commenting
  ```c
  /***/ I want to comment this testing section
  section_report();
  /* Handle the end of section stuff */
  dump_table();
  /***/ end of commented out section */
  ```

- What is wrong with this code?
- You can fix it by writing
  ```c
  ifdef DEBUG
  section_report();
  /* Handle the end of section stuff */
  dump_table();
  endif
  ```
Pointers Recap I

- A pointer is a variable in C that contains a memory location.
- Pointers are used in programs to access memory and manipulate addresses.
  - We have already seen it briefly in scanf() where usage was scanf("%d", &v);

Pointers Recap II

- Declaration
  - int *p;
  - This creates `p`, which is of type “pointer to int”
  - The legal range of values for any pointer always includes the special address 0 and a set of positive integers that are interpreted as machine addresses on the system
- & is used to “point to” the address of a variable
  - This is used to dereference a variable’s memory location
  - Officially - & is an operator that retrieves the memory address of a variable
Pointer Recap III

- Examples
  - `p = &i; // p has the memory location of i` 
    - `// therefore *p points to i`
  - `p = 0; // shows assignment of p to 0`
  - `p = NULL; // same as p = 0;`
  - `p = (int *) 1307; // p now has an absolute address in memory` 
    - `// We do this by using a cast`
    - `// This is typically not done, why?`

Pointer example

- Typical example

```c
int var; // Declare an integer var
int *p; // Declare p as a pointer to an integer

var = 4; // Set the value of var to be 4
p = &var; // Set p to be the address of var

printf ("%d", p); // Is this accurate?

*p = 5; // Sets the value of the thing p is pointing to, to 5
p = 5; // What will this do?
```
#include <stdio.h>

int main () {
    int var;       // Declare an integer var
    int *p;       // Declare p as a pointer to an integer
    var = 4;      // Set the value of var to be 4
    p = &var;     // Set p to be the address of var
    printf("The value of var is %d\n", var);
    printf("The value of pointer p is %d\n", p);
    return 0;
}

---

**Pointer Addressing/Dereferencing**

- `int a, b;`  
  - a 7  
  - b 7  
  - p ?  

- `int *p;`  
  - a 7  
  - b 7  
  - p ?

- `a = b = 7;`  
- `p = &a;`  
  - a 7  
  - b 7  
  - p

- `printf("%d\n", *p);`  // What is printed?

- `*p = 3;`  
- `printf("%d\n", a);`  // What is printed?
* and & relationship

- Simply put, the dereference operator (*) is the inverse of the address operator (&).

```c
double x, y, *p;

p = &x;
y = *p;

// Here, p is assigned to address of x. Then y is assigned to the value of object pointed to by p

y = *&x;
y = x;
// How do these two statements relate to the above two?
```

ptreexample1.c

```c
#include <stdio.h>

int main () {
    int thing_var;
    int *thing_ptr;

    thing_var = 2;
    printf("Thing %d\n", thing_var);

    thing_ptr = &thing_var;
    *thing_ptr = 3;
    printf("Thing %d\n", thing_var);

    printf("Thing %d\n", thing_ptr);
    return 0;
}
```
Multiple pointers can point to one location

```c
int something;

int *first_ptr;
int *second_ptr;

something = 1;

first_ptr = &something;
second_ptr = first_ptr;
```

Convincing?

<table>
<thead>
<tr>
<th>Declarations and Initializations</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 3, j = 6, *p = &amp;i, *q = &amp;j, *r; double x;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>p == &amp;i</td>
<td>p == (&amp; i)</td>
<td>1</td>
</tr>
<tr>
<td>p = i + 7</td>
<td>p = (i+7)</td>
<td>illegal</td>
</tr>
<tr>
<td>* * &amp; p</td>
<td><em>(</em> (&amp;p))</td>
<td>3</td>
</tr>
<tr>
<td>r = &amp;x</td>
<td>r = (&amp;x)</td>
<td>illegal</td>
</tr>
<tr>
<td>8 * * p / * q + 7</td>
<td>(((8 * (* p)) / (* q)) + 7)</td>
<td>11</td>
</tr>
<tr>
<td>*(r = &amp;j) == *p</td>
<td>(* (r = (&amp;j))) == (*p)</td>
<td>18</td>
</tr>
</tbody>
</table>
Call by reference

- Pointers can be used as function arguments
- We have been typically using call by value
- Remember the swap function

```c
#include <stdio.h>

int swap (int a, int b);
int main () {
  int x=3, y=7;
  printf("%d %d\n", x, y);
  swap (x,y);
  printf("%d %d\n", x, y);
  return 0;
}
```

```c
int swap (int a, int b) {
  int tmp;
  tmp=a;
  a=b;
  b=tmp;
  return a;    // I can return only one value, what do I return?
}
```

Call by reference II

- Note that the call-by-value has problems in that only the method’s local values are affected.
- Therefore we need something else
  - Pointers to the rescue
  - We call other functions and pass parameters by reference
  - New code looks like
Call by reference III

```c
#include <stdio.h>

int swap (int *, int *);

int main() {
    int x=3, y=7;
    printf("%d %d\n", x, y);
    swap (&x,&y);
    printf("%d %d\n", x, y);
    return 0;
}

int swap (int *p, int *q) {
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
} //ptrexample3.c
```

---

Call by reference IV

- Another example

```c
#include <stdio.h>

void inc_count (int *count_ptr)

int main () {
    int count = 0;
    while (count < 10)
        inc_count(&count);
    return 0;
}

void inc_count(int *count_ptr) {
    (*count_ptr)++;
}
```
ptrexample4.c

```c
#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;
    ptr = &my_array[0]; /* point our pointer to the first 
                         element of the array */
    printf("\n\n");
    for (i = 0; i < 6; i++)
    {
        printf("my_array[\%d] = \%d   ",i,my_array[i]); /*<-- A */
        printf("ptr + \%d = \%d\n",i, *(ptr + i));        /*<-- B */
    }
    return 0;
}
```

ptrexample5.c

```c
#include <stdio.h>

char strA[80] = "A string to be used for demonstration purposes";
char strB[80];

int main(void)
{
    char *pA; /* a pointer to type character */
    char *pB; /* another pointer to type character */
    puts(strA); /* show string A */
    pA = strA; /* point pA at string A */
    puts(pA); /* show what pA is pointing to */
    pB = strB; /* point pB at string B */
    putchar(\n); /* move down one line on the screen */
    while(*pA != \0) /* line A (see text) */
    {
        *pB++ = *pA++; /* line B (see text) */
    }
    *pB = \0; /* line C (see text) */
    puts(strB); /* show strB on screen */
    return 0;
}
```
ptrexample6.c

```c
#include <stdio.h>
#include <string.h>

struct tag {
    char lname[20];      /* last name */
    char fname[20];      /* first name */
    int age;             /* age */
    float rate;          /* e.g. 12.75 per hour */
};

struct tag my_struct;       /* declare the structure my_struct */

int main(void)
{
    strcpy(my_struct.lname,"Jensen");
    strcpy(my_struct.fname,"Ted");
    printf("%s\n",my_struct.fname);
    printf("%s\n",my_struct.lname);
    return 0;
}
```

Dynamic memory allocation

- void *malloc(size_t size);
- int array[10];
- /* Allocate space for an array with ten elements of type int. */
  int *ptr = malloc(10 * sizeof (int));
  if (ptr == NULL) {
      /* Memory could not be allocated, the program should handle the error here as appropriate. */
  }
  else {
      /* If ptr is not NULL, allocation succeeded. */
  }
```
Resize the allocated memory

- `void *realloc(void *pointer, size_t size);`
- To grow or shrink a block of memory
- Returns a pointer to some block in the memory that contains the same data as pointer

Memory leaks

- If you don’t use `free(void * pointer)` after `malloc` or `realloc`
- `void free(void *pointer)`
- Allocated memory won’t be released by itself until the program exits
example

```c
#include <stdio.h>
#include <stdlib.h> /* for malloc() function*/

int main()
{
    int *p;
    p = (int *)malloc(sizeof(int));
    if (p == 0)
    {
        printf("ERROR: Out of memory\n");
        return 1;
    }
    *p = 5;
    printf("Address of the location holding %d is 0x%p\n", *p, p);
    free(p);
    return 0;
}
```

Assignment

- Read Ch. 12~15 from the Practical C Programming book.
- HW#3