Pointers

Review

- recursion
  - scoping rule enforced by auto class
  - solution formation
- arrays and pointers

```
double  a[2], *p, *q;

p = a;              /* points to base of array */
q = p + 1;          /* equivalent to q = &a[1] */
printf("%d\n", q - p);                  /* 1 is printed */
printf("%d\n", (int) q - (int) p); /* 8 is printed */
```

- call-by-reference

Relation between Arrays and Pointers

- int a[10], i;
  - a[i] is equivalent to *(a + i)
- int i, *p
  - p[i] is equivalent to *(p + i)
  - a + i is equivalent to &a[i]

Arrays as Function Arguments

- When an array is passed as an argument to a function, the base address *value* is passed.
  - the array elements are not copied

- equivalent function headers

```
double sum(double a[], int n);
double sum(double *a, int n)
```

```
double sum(double a[], int n)
  /* n is the size of a[] */
{
    int   i;
    double sum = 0.0;

    for (i = 0; i < n; ++i)
       sum += a[i];
    return sum;
}
```

```
int a[] = {7, 3, 66, 3, -5, 22, 77, 2};
                bubble(a, 8);
```

```
void bubble(int a[], int n)
                        /* n is the size of a[] */
{
    int     i, j;
    void    swap(int *, int *);

    for (i = 0; i < n - 1; ++i)
        for (j = n - 1; j > i; --j)
            if (a[j-1] > a[j])
                swap(&a[j-1], &a[j]);
}
```

Example: Bubble Sort (very inefficient, for array of size n, the number of comparisons is proportional to n^2)

bubblesort.c

Dynamic Memory Allocation

- The standard C lib contains

  void * calloc(int n, int m)

  void * malloc(int m);

  – if failed, NULL is returned

- calloc (n, m) is equivalent to

  p = malloc (n*m)

  memset(p, 0, m*n);

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

int main(void)
{
   int  *a;                    /* to be used as an array */
   int  n;                     /* the size of the array  */
   scanf("%d", &n);
   a = calloc(n, sizeof(int));     /* get space for a */
   ...                                /* use a as an array */
   free(a);  /* release space occupied by an array a */
}
```

Memory Release

- You'd better free the allocated space
  - free(p);
  - p must be the pointer to the space allocated by calloc() or malloc()
- If you forget to free,
  - it will be freed when the process exits for some systems like Linux, Windows
  - for some other systems, nothing is guaranteed

Strings

- review
  - char *p = “abcde”;
  - char s[] = “abcde”;
  - char s[] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘₩0’};

```c
#include <ctype.h>

int word_cnt(char *s)
{
    int     cnt = 0;

    while (*s != '\0') {
        while (isspace(*s))       /* skip white space */
            ++s;
        if (*s != '\0') {             /* found a word */
            ++cnt;
            while (!isspace(*s) && *s != '\0')
                ++s;                  /* skip the word */
        }
    }
    return cnt;
}
```

String Functions

- ANSI C Lib contains many useful functions
  - char *strcat(char *s1, const char *s2);
    - result is in *s1
    - what if there is no space after s1?
  - int strcmp(const char *s1, const char *s2);
    - returns negative, zero, positive depending on the lexicographical order
  - char *strcpy(char *s1, const char *s2);
    - copy s2 to s1
    - what if s2 is longer than s1?
  - size_t strlen(const char *s);
    - size_t is usually unsigned int

```
unsigned strlen(const char *s)
{
    register int    n;

    for (n = 0; *s != '\0'; ++s)
        ++n;
    return n;
}
```

```
char   *strcat(char *s1, const char *s2)
{
    register char   *p = s1;

    while (*p)
        ++p;
    while (*p++ = *s2++)
        ;
    return s1;
}
```

Declarations and initializations

```
char    s1[] = "beautiful big sky country",
        s2[] = "how now brown cow";
```

Expression	Value
`strlen(s1)`	25
`strlen(s2+8)`	9
`strcmp(s1, s2)`	negative integer
Statements	**What gets printed**
`printf("%s", s1 + 10)`	big sky country
`strcpy(s1 + 10, s2 + 8)`	
`strcat(s1, "s!")`	
`printf("%s", s1)`	beautiful brown cows!

Multidimensional Arrays

- An array of arrays can be created
  - double a[3][7];
  - it is an array of three a[7]'s
  - the base address is &a[0][0], NOT a
- You can expand it to three dimensional arrays

int a[3][5]					
	Col 1	Col 2	Col 3	Col 4	Col 5
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Expression equivalent to a[i][j]
*(a[i] + j)
(*(a + i))[j]
(((a + i)) + j)
*(&a[0][0] + 5*i + j)

Initialization

Three equivalent initializations:

```
int  a[2][3] = {1, 2, 3, 4, 5, 6};
int  a[2][3] = {{1, 2, 3}, {4, 5, 6}};
int  a[][3] = {{1, 2, 3}, {4, 5, 6}};

int a[2][2][3] = {0};
/* all elements of a initialized to 0 */
```

Arrays of Pointers

- char *w[N];
  - an array of pointers
  - each pointer is to char

- ragged array
  - char *p[2] = {“abc”, “1234567890”};

  read the sort_words example in the textbook

Arguments to main( )

- argc and argv are used for main()
  - argc is the number of arguments
  - argv is an array of pointers
    - argv[0] is the name of the main program
    - then naturally, argc >= 1

my_echo.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int   i;

    printf("argc = %d\n", argc);
    for (i = 0; i < argc; ++i)
       printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

```
$ my_echo midterm is on Thursday
```

Functions as Arguments

- a function name can be passed as an argument
- think a function name as a pointer (like an array)
- (*f)(x)
  - f is a pointer to a function
  - *f is a function
  - (*f)(x) is call to the function
- if you are still confused, just follow the example

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

double    f(double);
double    sum_square(double (*)(double), int, int);
```

```c
#include "sum_sqr.h"

int main(void)
{
   printf("%s%.7f\n%s%.7f\n",
      " First computation: ", sum_square(f, 1, 10000),
      "Second computation: ", sum_square(sin, 2, 13));
   return 0;
}
```

```
double sum_square(double f(double), int m, int n)
{
    int       k;
    double    sum = 0.0;

    for (k = m; k <= n; ++k)
        sum += f(k) * f(k);
    return sum;
}


double f(double x)
{
    return 1.0 / x;
}
```

Functions as Arguments

- double g(double) returns double
- double *g(double) returns a pointer

- equivalent function prototype definitions

```
double sum_square(double f(double x), int m, int n);
double sum_square(double f(double), int m, int n);
double sum_square(double f(double), int, int);
double sum_square(double (*f)(double), int, int);
double sum_square(double (*)(double), int, int);
```

const volatile

- const int N = 3;
  - i cannot be changed after initialization
  - i cannot be used for array definition like
    - int k[N];

- extern const volatile int real_time_clock;
  - this variable is modified by other part of a computer,
  - but you cannot change the value, JUST READ it