

Arrays, Pointers and Strings

Chapter 6

One Dimensional Arrays

```
#define    N    100

int a[N];

for ( i = 0; i < N; ++i )
    sum += a[i];
```

Initializing Arrays

```
float f[5] = { 0.0, 1.0, 2.0, 3.0, 4.0 };
```

```
int a[100] = { 0 };
```

```
int a[] = { 2, 3, 5, -7 };
```

```
char s[] = "abc";
```

```
char s[] = { 'a', 'b', 'c', '\0' };
```

Pointers

```
p = 0;
```

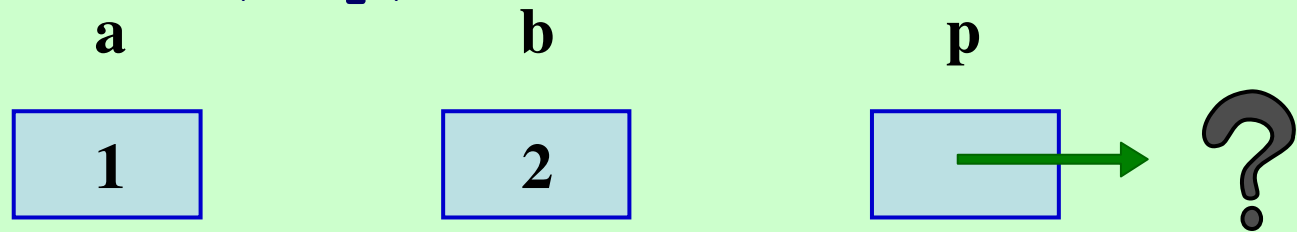
```
p = NULL;
```

```
p = &i;
```

```
p = (int *) 1776;
```

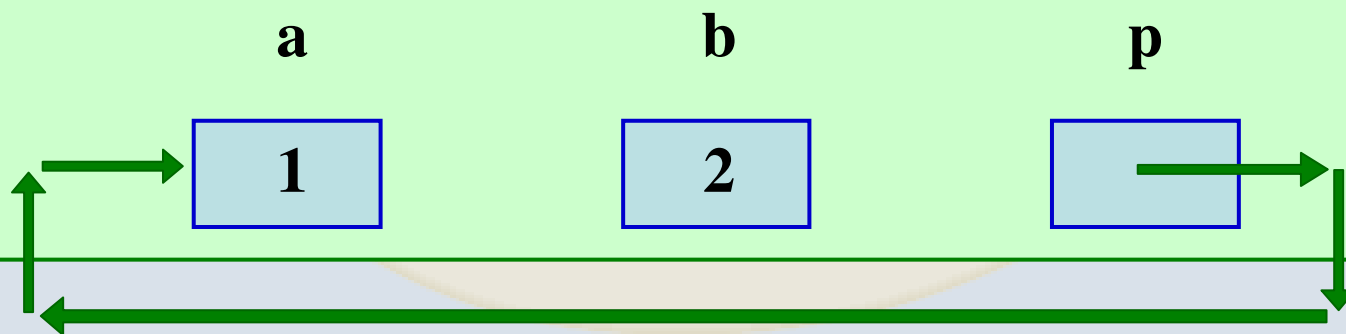
Pointers

```
int a = 1, b = 2, *p;
```



```
p = &a;
```

```
b = *p; is equivalent to b = a;
```



Pointers

```
int a = 1, b = 2, *p;
```

```
p = &a;
```

```
b = *p; /* is equivalent to b = a;*/
```

This is also correct:

```
int a = 1, b = 2, *p = &a;
```

Pointers

```
#include <stdio.h>

int main(void)
{
    int    i = 7, *p;
    p = &i;
    printf( "%s%d\n%s%u\n", "    Value of i: ", *p,
            "Location of i: ", p );
    return 0;
}
```

Printing result:

```
    Value of i: 7
Location of i: 251657504
```

Declarations and Initializations

```
int i = 3, j = 5, *p = &i, *q = &j, *r;  
double x;
```

Expression	Equivalent Expression	Value
<code>p == &i</code>	<code>p == (&i)</code>	1
<code>**&p</code>	<code>* (* (&p))</code>	3
<code>r = &x</code>	<code>r = (& x)</code>	illegal
<code>7 * * p / * q + 7</code>	<code>((7 * (*p)) / (*q)) + 7</code>	11

Declarations and Initializations

Declaration

```
int *p;  
float *q;  
void *v;
```

Legal Assignment	Illegal Assignment
p = 0;	p = 1;
p = (int*) 1;	v = 1;
p = v = q;	p = q;
p = (int*) q;	

More illegal:

```
&3,  
&(k + 99),  
register v;  
&v
```

Call by Reference

```
#include <stdio.h>
void swap(int *p, int *q)
{
    int    tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}

int main(void)
{
    int    i = 3, j = 5;
    swap(&i, &j);
    printf("%d %d\n", i, j );
    return 0;
}
```

The Relationship Between Arrays and Pointers

An array name is an address!

$a[i]$ is equivalent to **$*(a + i)$**

$p[i]$ is equivalent to **$*(p + i)$**

Pointer arithmetic

$p = a$ is equivalent to **$p = \&a[0]$**

$p = a + 1$ is equivalent to **$p = \&a[1]$**

Arrays and Pointers

```
for ( p = a; p < &a[ N ]; ++p )  
    sum += *p;
```

is equivalent to:

```
for ( i = 0; i < N; ++i )  
    sum += *( a + i );
```

is equivalent to:

```
p = a;  
for ( i = 0; i < N; ++i )  
    sum += p[ i ];
```

Illegal expressions

Illegal expressions

`a = p`

`++a`

`a += 2`

`&a`

Pointer Arithmetic and Element Size

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

p = a; /* points to a[0] */

q = p + 1; /* equivalent to q = &a[ 1 ] */

printf("%d\n", q - p);

printf("%d\n", (int)q - (int)p);
```

Arrays as Function Arguments

```
double sum(double a[], int n) /*n is the size of a*/
{
    int    i = 0;
    double sum = 0.0;

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

An alternative header

```
double sum(double *a, int n)
```

An Example: Bubble Sort

```
void bubble(int a[], int n )
{
    int i = 0, j = 0;
    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 ] );
}
```

Dynamic memory allocation

C supports `malloc()`, `calloc()`, `free()` for dynamic storage management, in the standard library. `malloc(n)` and `calloc()` are used to dynamically create space for arrays, structures and unions.

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

int main(void)
{
    int *a, n;
    ...
    a = calloc(n, sizeof(int));
    ...
}
```

Dynamic memory allocation - cont

`calloc(n, element_size)`

All bits are initialized to zero.

If the call is successful, a pointer of type `void*` is returned; otherwise, `NULL` is returned.

`malloc(n_bytes)`

allocates `size_t` bytes of memory, but without initialization.

`free(ptr)`

causes the memory pointed to by `ptr` to be deallocated.

If `ptr` is not `NULL` it must be the base address of memory allocated by `malloc()`, `calloc()` or `realloc()` that has not been deallocated by `free` or `realloc()`.

Dynamic memory allocation - example

```
#include "array.h"

int main(void) {
    int *a, n;
    srand(time(NULL)); /* seed random number generator */
    for ( ; ; ) {
        printf("Input n: ");
        if (scanf("%d", &n) != 1 || n < 1)
            break;
        putchar('\n');
        a = calloc(n, sizeof(int)); /* space for a*/
        fill_array(a, n);
        wrt_array(a, n);
        printf("sum = %d\n\n", sum_array(a, n));
        free(a);
    }
    return 0;
}
```

Merge

```
void merge(int a[], int b[], int c[], int m, int n)
{
    int    i = 0, j = 0, k = 0;
    while ( i < m && j < n )
        if ( a[i] < b[j] )
            c[k++] = a[i++];
        else
            c[k++] = b[j++];

    while ( i < m )
        c[k++] = a[i++];
    while ( j < n )
        c[k++] = b[j++];
}
```

Mergesort

```
#include <stdio.h>
#include <stdlib.h>
void merge(int *, int *, int *, int, int);

void mergesort(int key[], int n)
{
    int j = 0, k = 0, m = 0, *w = NULL;
    for ( m = 1; m < n; m *= 2 ) ;

    if ( m != n )
    {
        printf("ERROR: array Size of is not a power of
2 - bye!\n" );
        exit( 1 );
    }
}
```

Mergesort

```
w = calloc( n, sizeof( int ) );
assert( w != NULL);
for ( k = 1; k < n; k *= 2 )
{
    for ( j = 0; j < n - k; j += 2 * k )
        merge( key + j, key + j + k, w + j, k, k );

    for (j = 0; j < n; ++j)
        key[j] = w[j];
}
free(w);
w = NULL;
}
```

Mergesort

```
#include <stdio.h>
#define KEYSIZE 16
void mergesort(int *, int);

int main(void)
{
    int i, key[] = { 4, 3, 1, 67, 55, 8, 0, 4, -5,
                    37, 7, 4, 2, 9, 1, -1 };

    mergesort(key, KEYSIZE);
    printf("After mergesort:\n" );
    for ( i = 0; i < KEYSIZE; ++i )
        printf("%4d", key[i] );
    putchar( '\n' );
    return 0;
}
```

Strings

```
char *p = "abc"
```

```
printf("%s %s\n", p, p + 1 );
```

```
char s[] = "abcde";
```

is equivalent to:

```
char s[] = {'a', 'b', 'c', 'd', 'e', '\0'};
```

Strings

`char *p = "abcde"` vs. `char s[] = "abcde"`

p



s



Strings

"abc"[1] and *("abc" + 2) make sense

```
char *s = NULL;
int  nfrogs = 0;
. . .
s = (nfrogs == 1)? "" : "s";
printf("we found %d frog%s in the pond!\n", nfrogs,
      s );
```

Count the number of words in a string

```
#include <ctype.h>
int word_cnt(char *s)
{
    int cnt = 0;
    while (*s != '\0')
    {
        while (isspace(*s))
            ++s;
        if (*s != '\0')
        {
            ++cnt;
            while (!isspace(*s) && *s != '\0')
                ++s;
        }
    }
    return cnt;
}
```

String Handling Functions in the Standard Library

```
char *strcat(char *s1, const char *s2);
```

```
int strcmp(const char *s1, const char *s2);
```

```
char *strcpy(char *s1, const char *s2);
```

```
unsigned strlen(const char *s);
```

strlen implementation

```
unsigned strlen ( const char *s )
{
    register int    n = 0;

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

strcat implementation

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

    while ( *p )
        ++p;

    while ( *p++ = *s2++ ) ;

    return s1;
}
```

Multidimensional Arrays

```
int sum( int a[][5] )
{
    int    i = 0, j = 0, sum = 0;

    for ( i = 0; i < 3; ++i )
        for ( j = 0; j < 5; ++j )
            sum += a[i][j];
    return sum;
}
```

Arrays of Pointers - Sort Words Lexicographically

Includes and function templates

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXWORD 50
#define N 1000

void sort_words(char *[], int);
void swap(char **, char **);
```

Sort Words Lexicographically

```
int main(void)
{
    char    *w[N];
    char    word[MAXWORD];
    int     n = 0, i = 0;
    for ( i = 0; scanf("%s", word) == 1; ++i )
    {
        assert ( i < N );
        w[i] = calloc(strlen(word) + 1, sizeof(char));
        assert(w[i] != NULL); /* simplified test */
        strcpy(w[i], word);
    }
    n = i;
    sort_words( w, n );
    for ( i = 0; i < n; ++i )
        printf("%s\n", w[i] );
    return 0;
}
```

Sort Words Lexicographically

```
void sort_words( char *w[], int n )
{
    int    i = 0, j = 0;
    for ( i = 0; i < n; ++i )
        for ( j = i + 1; j < n; ++j )
            if ( strcmp(w[i], w[j]) > 0 )
                swap( &w[i], &w[j] );
}
```

```
void swap( char **p, char **q )
{
    char    *temp = NULL;
    temp = *p;
    *p = *q;
    *q = temp;
}
```

Arguments to main()

Two arguments, conventionally called `argc` and `argv` can be used with `main` to communicate with OS.

`argc` is the number of command line arguments.

`argv` is an array of pointers to char.

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

```
void main(int argc, char *argv[])
```

```
{
```

```
    int    i = 0;
```

```
    printf("argc = %d\n", argc );
```

```
    for ( i = 0; i < argc; ++i )
```

```
        printf("argv[%d] = %s\n", i, argv[i] );
```

```
}
```

Arguments to main() example

if we put the executable code in file *demo_echo.exe* and give the command

```
> demo_echo I like 2 program
```

We get

```
argc = 5
```

```
argv[0] = demo echo
```

```
argv[1] = I
```

```
argv[2] = like
```

```
argv[3] = 2
```

```
argv[4] = program
```

```
#include <stdio.h>

void main(int argc, char *argv[])
{
    int    i = 0;

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

Ragged Arrays

```
#include <stdio.h>
int main(void)
{ char    a[2][15]= { "abc:", "a is for apple" };
  char    *p[2]    = {"abc:", "a is for apple"};

  printf("%c%c%c %s %s\n%c%c%c %s %s\n",
         a[0][0], a[0][1], a[0][2], a[0], a[1],
         p[0][0], p[0][1], p[0][2], p[0], p[1] );
  return 0;
}
```

abc abc: a is for apple

abc abc: a is for apple

Functions as Arguments

```
double sum_square( double f(double), int m, int n )
{
    int      k = 0;
    double   sum = 0.0;

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

```
double sum_square( double (*f)(double), int m, int n )
{
    .....
```

Functions as Arguments

```
#include <stdio.h>
double  f(double), sin(double), sum_square(double
    (*)(double), int, int);

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

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

Find a root of $f()$ by the bisection method

```
#define EPS 1e-12

double root(double f(double), double a, double b)
{
    double m = (a + b) / 2.0; /* midpoint */

    if (f(m) == 0.0 || b - a < EPS)
        return m;
    else if (f(a) * f(m) < 0.0)
        return root(f, a, m);
    else return root(f, m, b);
}
```

Find a root of $f()$ by the bisection method

```
#include <stdio.h>
double p(double);
double root(double (*)(double), double, double);

int main(void)
{
    double x = root( p, 0.0, 3.0 );
    printf("%s%.16f\n%s%.16f\n", "Approximate root: ", x,
           " Function value: ", p( x ) );
    return 0;
}
```

Find a root of $f()$ by the bisection method

```
double p(double x)
{
    return (x * x * x * x * x - 7.0 * x - 3.0);
}
```

Approximate root: 1.7196280914844584

Function value: 0.00000000000000317