ARRAYS & HEAP

Content

In this chapter, you will learn:

- To introduce the array data structure
- To understand the use of arrays
- To understand how to define an array, initialize an array and refer to individual elements of an array
- To be able to pass arrays to functions
- To be able to define and manipulate multi-dimensional arrays
- To be able to use arrays as pointers
- To be able to create variable-length arrays whose size is determined at execution time

Introduction

Arrays

- Structures of related data items
- Static entity – same size throughout program
- Dynamic data structures will be discussed later

Arrays

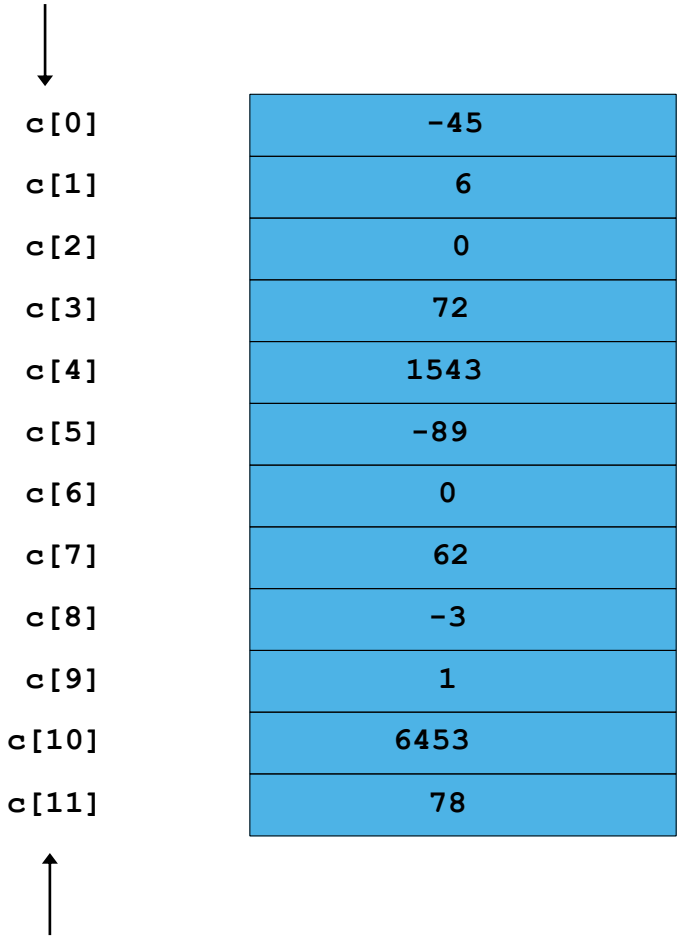
- Array
  - Group of consecutive memory locations
  - Same name and type

- To refer to an element, specify
  - Array name
  - Position number

- Format:

  arrayname`[` *position number* `]`
  - First element at position 0
  - n element array named `c:`
    - `c[ 0 ], c[ 1 ]...c[ n - 1 ]`

Name of array (Note that all elements of this array have the same name, **c**)



Position number of the element within array **c**

Arrays

- Array elements are like normal variables

```
c[ 0 ] =  3;
printf( "%d", c[ 0 ] );
```

  - Perform operations in subscript. If x equals 3

    c[5 - 2] == c[3] == c[x]

    c[x+1] == c[4]

    c[x-1] == c[2]

Arrays

Operators						Associativity	Type
[]	()					left to right	highest
++	--	!	(*type*)			right to left	unary
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
&&						left to right	logical and
||						left to right	logical or
?:						right to left	conditional
=	+=	-=	*=	/=	%=	right to left	assignment
,						left to right	comma

Fig. 6.2 Operator precedence.

double x[8];

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

i=5

printf("%d %.1f", 4, x[4]);	4 2.5
printf("%d %.1f", i, x[i]);	5 12.0
printf("%.1f", x[i]+1);	13.0
printf("%.1f", x[i]+i);	17.0
printf("%.1f", x[i+1]);	14.0
printf("%.1f", x[i+i]);	invalid
printf("%.1f", x[2*i]);	invalid
printf("%.1f", x[2*i-3]);	-54.5
printf("%.1f", x[(int)x[4]]);	6.0
printf("%.1f", x[i++]);	12.0
printf("%.1f", x[--i]);	12.0

May result in a run-time error
Display incorrect results

double x[8];

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

i=5

x[i-1] = x[i]

x[i] = x[i+1]

x[i]-1 = x[i] Illegal assignment statement!

Defining Arrays

- When defining arrays, specify
  - Type of array
  - Name
  - Number of elements

```
arrayType arrayName[ numberOfElements ];
```

Examples:

```
int c[ 10 ];
float myArray[ 3284 ];
```

- Defining multiple arrays of same type
  - Format similar to regular variables
  - Example:

```
int b[ 100 ], x[ 27 ];
```

Examples Using Arrays

- Initializers

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

  - If not enough initializers, elements with missing values become 0

```
int n[ 5 ] = { 0 }
```

    All elements 0

  - C arrays have no bounds checking

- If size omitted, initializers determine it

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

  - 5 initializers, therefore 5 element array

Initializing an Array

```
1  /* Fig. 6.3: fig06_03.c
2     initializing an array */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     int n[ 10 ]; /* n is an array of 10 integers */
9     int i;       /* counter */
10
11    /* initialize elements of array n to 0 */
12    for ( i = 0; i < 10; i++ ) {
13       n[ i ] = 0; /* set element at location i to 0 */
14    } /* end for */
15
16    printf( "%s%13s\n", "Element", "Value" );
17
18    /* output contents of array n in tabular format */
19    for ( i = 0; i < 10; i++ ) {
20       printf( "%7d%13d\n", i, n[ i ] );
21    } /* end for */
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
```

Program Output

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

```c
/* Fig. 6.4: fig06_04.c
   Initializing an array with an initializer list */
#include <stdio.h>

/* function main begins program execution */
int main()
{
   /* use initializer list to initialize array n */
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
   int i; /* counter */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

Program Output

Element	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

```c
/* Fig. 6.5: fig06_05.c
   Initialize the elements of array s to the even integers from 2 to 20 */
#include <stdio.h>
#define SIZE 10

/* function main begins program execution */
int main()
{
   /* symbolic constant SIZE can be used to specify array size */
   int s[ SIZE ]; /* array s has 10 elements */
   int j;         /* counter */

   for ( j = 0; j < SIZE; j++ ) { /* set the values */
      s[ j ] = 2 + 2 * j;
   } /* end for */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array s in tabular format */
   for ( j = 0; j < SIZE; j++ ) {
      printf( "%7d%13d\n", j, s[ j ] );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

Program Output

Element	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

Examples

- Reading values into an array

```
int i, x[100];
```

```
for (i=0; i < 100; i=i+1)
{
  printf("Enter an integer: ");
  scanf("%d",&x[i]);
}
```

- Summing up all elements in an array

```
int sum = 0;
for (i=0; i<=99; i=i+1)
   sum = sum + x[i];
```

Examples

- Finding the location of a given value (`item`) in an array.

```
i = 0;
while ((i<100) && (x[i] != item))
   i = i + 1;

if (i == 100)
   loc = -1; // not found
else
   loc = i;  // found in location i
```

Examples

- Shifting the elements of an array to the left.

```
/* store the value of the first element in a
 * temporary variable
 */
temp = x[0];

for (i=0; i < 99; i=i+1)
   x[i] = x[i+1];

//The value stored in temp is going to be
the value of the last element:
x[99] = temp;
```

```c
/* Fig. 6.6: fig06_06.c
   Compute the sum of the elements of the array */
#include <stdio.h>
#define SIZE 12

/* function main begins program execution */
int main()
{
   /* use initializer list to initialize array */
   int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
   int i;          /* counter */
   int total = 0; /* sum of array */

   /* sum contents of array a */
   for ( i = 0; i < SIZE; i++ ) {
      total += a[ i ];
   } /* end for */

   printf( "Total of array element values is %d\n", total );

   return 0; /* indicates successful termination */

} /* end main */
```

Total of array element values is 383

```c
/* Fig. 6.7: fig06_07.c
   Student poll program */
#include <stdio.h>
#define RESPONSE_SIZE 40 /* define array sizes */
#define FREQUENCY_SIZE 11

/* function main begins program execution */
int main()
{
   int answer; /* counter */
   int rating; /* counter */

   /* initialize frequency counters to 0 */
   int frequency[ FREQUENCY_SIZE ] = { 0 };

   /* place survey responses in array responses */
   int responses[ RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
        1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
        5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };

```

```
21    /* for each answer, select value of an element of array responses
22         and use that value as subscript in array frequency to
23         determine element to increment */
24    for ( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
25       ++frequency[ responses [ answer ] ];
26    } /* end for */
27
28    /* display results */
29    printf( "%s%17s\n", "Rating", "Frequency" );
30
31    /* output frequencies in tabular format */
32    for ( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
33       printf( "%6d%17d\n", rating, frequency[ rating ] );
34    } /* end for */
35
36    return 0; /* indicates successful termination */
37
38 } /* end main */
```

Rating	Frequency
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

```c
/* Histogram printing program */

#include <stdio.h>
#define SIZE 10

int main()
{
   int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
   int i, j;

   printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );

   for ( i = 0; i <= SIZE - 1; i++ ) {
      printf( "%7d%13d          ", i, n[i]) ;

      for ( j = 1; j <= n[ i ]; j++ )   /* print one bar */
         printf( "%c", '*' );

      printf( "\n" );
   }

   return 0;
}
```

Program Output

```
Element         Value       Histogram
      0            19       *******************
      1             3       ***
      2            15       ***************
      3             7       *******
      4            11       ***********
      5             9       *********
      6            13       *************
      7             5       *****
      8            17       *****************
      9             1       *
```

```c
/* Fig. 6.9: fig06_09.c
   Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

/* function main begins program execution */
int main()
{
   int face;                      /* random number with value 1 - 6 */
   int roll;                      /* roll counter */
   int frequency[ SIZE ] = { 0 }; /* initialize array to 0 */

   srand( time( NULL ) ); /* seed random-number generator */

   /* roll die 6000 times */
   for ( roll = 1; roll <= 6000; roll++ ) {
      face = rand() % 6 + 1;
      ++frequency[ face ]; /* replaces 26-line switch of Fig. 5.8 */
   } /* end for */

   printf( "%s%17s\n", "Face", "Frequency" );

```

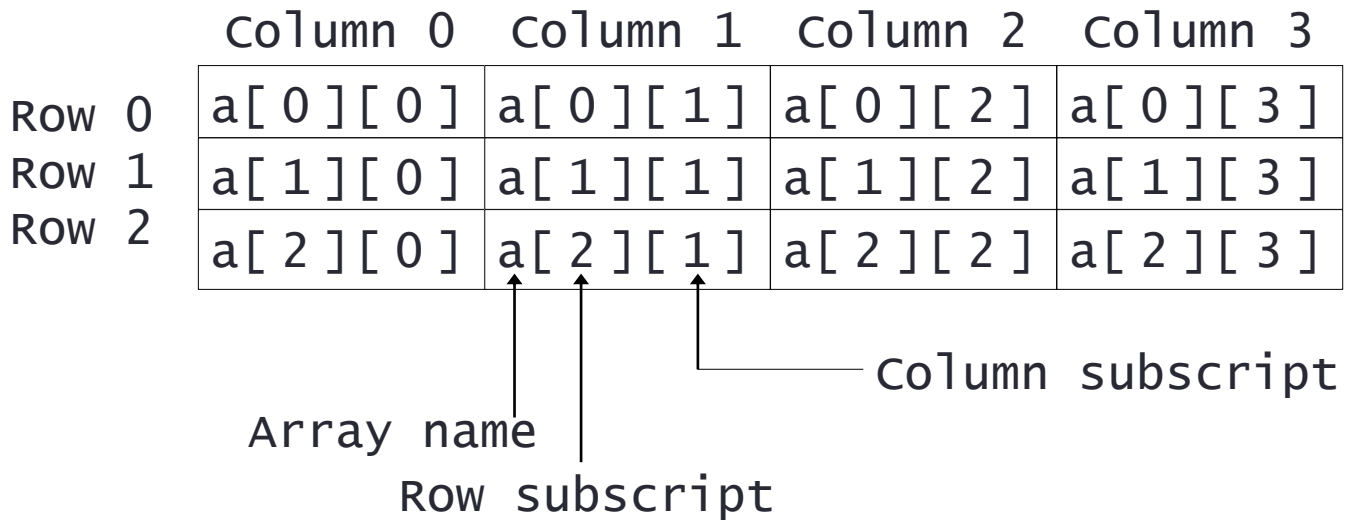
```
25    /* output frequency elements 1-6 in tabular format */
26    for ( face = 1; face < SIZE; face++ ) {
27       printf( "%4d%17d\n", face, frequency[ face ] );
28    } /* end for */
29
30    return 0; /* indicates successful termination */
31
32 } /* end main */
```

Program Output

```
Face          Frequency
   1               1029
   2                951
   3                987
   4               1033
   5               1010
   6                990
```

Multi-Dimensional Arrays

- Multiple subscripted arrays
  - Tables with rows and columns (m by n array)
  - Like matrices: specify row, then column



Multi-Dimensional Arrays

- Initialization
  - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
  - Initializers grouped by row in braces
  - If not enough, unspecified elements set to zero

    `int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };`

1	2
3	4

1	0
3	4

- Referencing elements
  - Specify row, then column

    `printf( "%d", b[ 0 ][ 1 ] );`

Example:Multi-Dimensional Array

```
#include <stdio.h>
int main()
{
   int i; /* counter */
   int j; /* counter */

   /* initialize array1, array2, array3 */
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
   int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   printf( "Values in array1 by row are:\n" );

   for ( i = 0; i <= 1; i++ ) {                          /* loop through rows */
            for ( j = 0; j <= 2; j++ )
                     printf( "%d ", array1[ i ][ j ] );  /* output column values */
            printf( "\n" );
   }
```

Example:Multi-Dimensional Array

```
   printf( "Values in array2 by row are:\n" );
   for ( i = 0; i <= 1; i++ ) {                    /* loop through rows */
            for ( j = 0; j <= 2; j++ )
                     printf( "%d ", array2[ i ][ j ] );    /* output column values */
            printf( "\n" );
   }

   printf( "Values in array3 by row are:\n" );
   for ( i = 0; i <= 1; i++ ) {                    /* loop through rows */
            for ( j = 0; j <= 2; j++ )
                     printf( "%d ", array3[ i ][ j ] );
            printf( "\n" );
   }

   return 0;
}
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Pointers and Arrays

- Arrays are implemented as pointers.
- Consider:

  `double list[3];`

  `&list[1]` : is the address of the second element

  `&list[i]` : the address of `list[i]` which is calculated by the formula

  base address of the array + i * *8*

The Relationship between Pointers and Arrays

- Arrays and pointers are closely related
  - Array name is like a constant pointer
  - Pointers can do array subscripting operations
- Declare an array **`b[ 5 ]`** and a pointer **`bPtr`**
  - To set them equal to one another use:

    ```
    bPtr = b;
    ```

    - The array name (**`b`**) is actually the address of first element of the array **`b[ 5 ]`**

      ```
      bPtr = &b[ 0 ]
      ```

    - Explicitly assigns **`bPtr`** to address of first element of **`b`**

The Relationship between Pointers and Arrays

- Element `b[ 3 ]`
  - Can be accessed by `*( bPtr + 3 )`
    - Where `n` is the offset. Called pointer/offset notation
  - Can be accessed by `bptr[ 3 ]`
    - Called pointer/subscript notation
    - `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself
    `*( b + 3 )`

Example (cont.)

```
/* Using subscripting and pointer notations with
 arrays */
#include <stdio.h>
int main(void)
{
 int i, offset, b[4]={10,20,30,40};
 int *bPtr = b;

/* Array is printed with array subscript notation */

 for (i=0; i < 4; i++)
    printf("b[%d] = %d\n", i, b[i]);
```

Example (cont.)

```
/* Pointer/offset notation where the pointer is
   the array name */

 for (offset=0; offset < 4; offset++)
     printf("*(b + %d) = %d\n",offset,*(b + offset));

/* Pointer subscript notation */
 for (i=0; i < 4; i++)
    printf("bPtr[%d] = %d\n", i, bPtr[i]);

/* Pointer offset notation */
 for (offset = 0; offset < 4; offset++)
    printf("*(bPtr + %d) = %d\n", offset"
                          "*(bPtr + offset)");

 return 0;
}
```

Example (cont.)

```
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

Passing Arrays as Parameters

- Arrays are passed to functions ***by reference***
  - the called functions can modify the element values in the callers' original arrays
  - individual array elements are passed ***by value*** exactly as simple variables are
- The function's parameter list must specify that an array will be received
- `int SumIntegerArray(int a[], int n)`
- It indicates that `SumIntegerArray` expects to receive an array of integers in parameter `a` and the number of array elements in parameter `n`
- The size of the array is not required between the array brackets

Passing Arrays as Parameters

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

Assume

```
int sum, list[5];
```

are declared in the main function. We can make the following function call:

```
sum = SumIntegerArray(list, 5);
```

Strings

- Character arrays
  - String “`first`” is really a static array of characters
  - Character arrays can be initialized using string literals
    - char string1[] = "first";
    - Null character `'\0'` terminates strings
    - `string1` actually has 6 elements
      - equivalent to `char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };`
  - Can access individual characters

    `string1[ 3 ] is character ‘s’`
  - Array name is address of array, so & not needed for scanf

    `scanf( "%s", string2 );`
    - Reads characters until whitespace encountered
    - Can write beyond end of array, be careful

```c
/* Fig. 6.10: fig06_10.c
   Treating character arrays as strings */
#include <stdio.h>

/* function main begins program execution */
int main()
{
   char string1[ 20 ];                 /* reserves 20 characters */
   char string2[] = "string literal"; /* reserves 15 characters */
   int i;                              /* counter */

   /* read string from user into array string2 */
   printf("Enter a string: ");
   scanf( "%s", string1 );

   /* output strings */
   printf( "string1 is: %s\nstring2 is: %s\n"
           "string1 with spaces between characters is:\n",
           string1, string2 );

   /* output characters until null character is reached */
   for ( i = 0; string1[ i ] != '\0'; i++ ) {
      printf( "%c ", string1[ i ] );
   } /* end for */
   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */
```

Program Output

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Example: String Copy

```
/* Copying a string using array notation and pointer notation. */
#include <stdio.h>
void copy1( char *s1, const char *s2 );
void copy2( char *s1, const char *s2 );

int main()
{
  char string1[ 10 ];           /* create array string1 */
  char *string2 = "Hello";      /* create a pointer to a string */
  char string3[ 10 ];           /* create array string3 */
  char string4[] = "Good Bye"; /* create a pointer to a string */

  copy1( string1, string2 );
  printf( "string1 = %s\n", string1 );
  copy2( string3, string4 );
  printf( "string3 = %s\n", string3 );

  return 0;
}
```

Example

```
/* copy s2 to s1 using array notation */
   void copy1( char *s1, const char *s2 )
   {
      int i;
      for ( i = 0; s2[ i ] != '\0'; i++ )
         s1[ i ] = s2[ i ];
      s1[ i ] = NULL;
   } /* end function copy1 */

   /* copy s2 to s1 using pointer notation */
   void copy2( char *s1, const char *s2 )
   {
      /* loop through strings */
      for ( ; *s2 != '\0'; s1++, s2++ )
         *s1 = *s2;
      *s1 = NULL;
   } /* end function copy2 */
```

Program Output

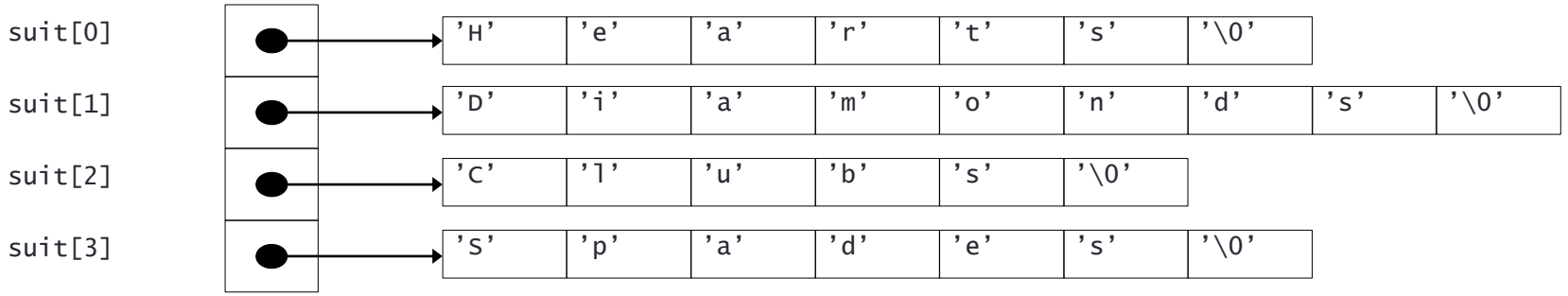
```
string1 = Hello
string3 = Good Bye
```

Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

```
char *suit[ 4 ] = { "Hearts", "Diamonds",
  "Clubs", "Spades" };
```

  - Strings are pointers to the first character
  - `char *` – each element of `suit` is a pointer to a `char`
  - The strings are not actually stored in the array `suit`, only pointers to the strings are stored



  - `suit` array has a fixed size, but strings can be of any size

`sizeof` function

- `sizeof`
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if `sizeof( int )` equals 4 bytes, then

```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

    - will print 40
- `sizeof` can be used with
  - Variable names
  - Type name
  - Constant values

Example

```
/* sizeof operator when used on an array name returns the number of
   bytes in the array. */
#include <stdio.h>
size_t getSize( float *ptr ); /* prototype */

int main(){
  float array[ 20 ]; /* create array */

  printf( "The number of bytes in the array is %d"
          "\nThe number of bytes returned by getSize is %d\n",
          sizeof( array ), getSize( array ) );

  return 0;
}

size_t getSize( float *ptr ) {
  return sizeof( ptr );
}
```

Program Output

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

Example

```
/* Demonstrating the sizeof operator */
#include <stdio.h>

int main()
{
   char c;            /* define c */
   short s;           /* define s */
   int i;             /* define i */
   long l;            /* define l */
   float f;           /* define f */
   double d;          /* define d */
   long double ld;    /* define ld */
   int array[ 20 ];   /* initialize array */
   int *ptr = array; /* create pointer to array */
```

Example

```
   printf( "       sizeof c = %d\tsizeof(char)  = %d"
           "\n     sizeof s = %d\tsizeof(short) = %d"
           "\n     sizeof i = %d\tsizeof(int) = %d"
           "\n     sizeof l = %d\tsizeof(long) = %d"
           "\n     sizeof f = %d\tsizeof(float) = %d"
           "\n     sizeof d = %d\tsizeof(double) = %d"
           "\n    sizeof ld = %d\tsizeof(long double) = %d"
           "\n sizeof array = %d"
           "\n   sizeof ptr = %d\n",
          sizeof c, sizeof( char ), sizeof s,
          sizeof( short ), sizeof i, sizeof( int ),
          sizeof l, sizeof( long ), sizeof f,
          sizeof( float ), sizeof d, sizeof( double ),
          sizeof ld, sizeof( long double ),
          sizeof array, sizeof ptr );

   return 0;
}
```

Example

Program Output

```
sizeof c = 1        sizeof(char)  = 1
sizeof s = 2        sizeof(short) = 2
sizeof i = 4        sizeof(int) = 4
sizeof l = 4        sizeof(long) = 4
sizeof f = 4        sizeof(float) = 4
sizeof d = 8        sizeof(double) = 8
sizeof ld = 8       sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Dynamic Memory Management

- **Static memory allocation:** space for the object is provided in the binary at compile-time
- **Dynamic memory allocation:** blocks of memory of arbitrary size can be requested at run-time

- The four dynamic memory management functions are `malloc`, `calloc`, `realloc`, and `free`.
- These functions are included in the header file `<stdlib.h>`.

Dynamic Memory Management

- `void *malloc(size_t size);`

- allocates storage for an object whose size is specified by `size`:
  - It returns a pointer to the allocated storage,
  - `NULL` if it is not possible to allocate the storage requested.
  - The allocated storage is not initialized in any way.

- **e.g.** `float *fp, fa[10];`

  ```
  fp = (float *) malloc(sizeof(fa));
  ```

  allocates the storage to hold an array of 10 floating-point elements, and assigns the pointer to this storage to fp.

Dynamic Memory Management

- `void *calloc(size_t nobj, size_t size);`

- allocates the storage for an array of `nobj` objects, each of size `size`.
  - It returns a pointer to the allocated storage,
  - `NULL` if it is not possible to allocate the storage requested.
  - The allocated storage is initialized to zeros.

- **e.g.** `double *dp, da[10];`

  `dp=(double *) calloc(10,sizeof(double));`

  allocates the storage to hold an array of 10 `double` values, and assigns the pointer to this storage to dp.

Dynamic Memory Management

- `void *realloc(void *p, size_t size);`

- changes the size of the object pointed to by `p` to `size`.
  - It returns a pointer to the new storage,
  - `NULL` if it is not possible to resize the object, in which case the object (`*p`) remains unchanged.
  - The new size may be larger (the original contents are preserved and the remaining space is unitialized) or smaller (the contents are unchanged upto the new size) than the original size.

Dynamic Memory Management

- **e.g.**
```
char *cp;
cp =(char *) malloc(sizeof("computer"));
strcpy(cp, "computer");
```

`cp` points to an array of 9 characters containing the null-terminated string `computer`.

```
cp = (char *) realloc(cp, sizeof("compute"));
```

discards the trailing '\0' and makes cp point to an array if 8 characters containing the characters in `computer`

```
cp=(char *)realloc(cp,sizeof("computerization"));
```

`cp` points to an array of 16 characters, the first 9 of which contain the null-terminated string `computer` and the remaining 7 are uninitialized.

Dynamic Memory Management

- `void *free(void *p);`

- deallocates the storage pointed to by p, where p is a pointer to the storage previously allocated by malloc, calloc, or realloc.

- **e.g.**
```
free(fp);
free(dp);
free(cp);
```

Dynamic Memory Management

- When using dynamic memory allocation, test for a `NULL` pointer return value, which indicates unsuccessful operation
- `free` dynamically allocated memory when it is no longer needed to avoid **memory leaks**
- Then set the pointer to `NULL` to eliminate possible further accesses to that memory
  - Referring to memory that has been deallocated is an error that typically results in the program crashing
- Trying to `free` memory not allocated dynamically is an error

Example

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

int main(void) {
  int *array, *p;
  int i,no_elements;
  printf("Enter number of elements: ");
  scanf("%d",&no_elements);

  printf("Enter the elements: ");
  array = ( int* )malloc( no_elements*sizeof( int ) );
  for(p=array,i=0; i<no_elements; i++, p++)
       scanf("%d",p);

  printf("Elements: ");
  for(p=array,i=0; i<no_elements; i++, p++)
       printf("%d ",*p);
  printf("\n");
```

Example

```
  array = ( int* )realloc(array, (no_elements+2)*sizeof( int ) );

  printf("Enter two new elements: ");
  for(p=array,i=0; i<no_elements; i++, p++) ;

  for(; i<no_elements+2; i++, p++)
       scanf("%d",p);

  printf("Elements: ");
  for(p=array,i=0; i<no_elements+2; i++, p++)
       printf("%d ",*p);
  printf("\n");

  free(array);
  return 0;
}
```

Program Output

```
Enter number of elements: 4
Enter the elements: 2 3 4 5
Elements: 2 3 4 5
Enter two new elements: 6 7
Elements: 2 3 4 5 6 7
```

Example: Dynamic Allocation of 2D Array (1)

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

#define ROW 3
#define COLUMN 4

/* Using a single pointer */
int main()
{
  int *arr = (int *)malloc(ROW * COLUMN * sizeof(int));

  int i, j, count = 0;
  for (i = 0; i <  ROW; i++)
    for (j = 0; j < COLUMN; j++)
      *(arr + i*COLUMN + j) = ++count;

  for (i = 0; i <  ROW; i++)
    for (j = 0; j < COLUMN; j++)
      printf("%d ", *(arr + i*COLUMN + j));

  return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Example: Dynamic Allocation of 2D Array (2)

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

#define ROW 3
#define COLUMN 4

/* Using an array of pointers */
int main()
{
  int i, j, count;

  int *arr[ROW];
  for (i=0; i<ROW; i++)
    arr[i] = (int *)malloc(COLUMN * sizeof(int));

  // Note that arr[i][j] is same as *(*(arr+i)+j)
  count = 0;
  for (i = 0; i <  ROW; i++)
   for (j = 0; j < COLUMN; j++)
     arr[i][j] = ++count; // OR *(*(arr+i)+j) = ++count

  for (i = 0; i <  ROW; i++)
   for (j = 0; j < COLUMN; j++)
     printf("%d ", arr[i][j]);

 return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Example: Dynamic Allocation of 2D Array (3)

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

#define ROW 3
#define COLUMN 4

/* Using pointer to a pointer */
int main()
{
  int i, j, count;

  int **arr = (int **)malloc(ROW * sizeof(int *));
  for (i=0; i<ROW; i++)
     arr[i] = (int *)malloc(COLUMN * sizeof(int));

  // Note that arr[i][j] is same as *(*(arr+i)+j)
  count = 0;
  for (i = 0; i <  ROW; i++)
   for (j = 0; j < COLUMN; j++)
     arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count

  for (i = 0; i <  ROW; i++)
   for (j = 0; j < COLUMN; j++)
     printf("%d ", arr[i][j]);

 return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Example: Dynamic Allocation of 2D Array (4)

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

#define ROW 3
#define COLUMN 4

/* Using double pointer and one malloc call for all rows */
int main()
{
  int **arr;
  int count = 0,i,j;

  arr  = (int **)malloc(sizeof(int *) * ROW);
  arr[0] = (int *)malloc(sizeof(int) * COLUMN * ROW);

  for(i = 0; i < ROW; i++)
    arr[i] = (*arr + COLUMN * i);

  for (i = 0; i < ROW; i++)
    for (j = 0; j < COLUMN; j++)
      arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count

  for (i = 0; i <  ROW; i++)
    for (j = 0; j < COLUMN; j++)
      printf("%d ", arr[i][j]);

  return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10 11 12
```