5 Arrays and Pointers

5.1 One-dimensional arrays

Arrays offer a convenient way to store and access blocks of data. Think of arrays as a sequential list that offers indexed access. For example, a list of quiz scores of this C programming course with 110 students may be stored in a C array. The idea is that instead of having 110 different identifiers, one for each student, we can use a single identifier for the entire class: the array `grade[]`.

A particular user's score could be accessed by a C program using the construction:

```
grade[index]
```

where `index` would be an integer, in this case in the interval $[0, 109]$. Note that the first student would be accessed using the index 0 and the 110^{th} student by the index 109!

A one-dimensional array in C is therefore a list of variables that are all of the *same type* and are referenced through a common name. An individual variable in the array is called an *array element*.

An one-dimensional array is declared using the syntax:

```
type variable_name[size];
```

where size is an integer. One may also declare a variable to be an array with no storage allocated at declaration time. We will study those later.

For example,

```
int my_array[50];
double velocity[100];
```

declares an array of integers `my_array` with room for 50 elements and an array of doubles `velocity` with room for 100 elements.

We repeat one more time that that, in C, array subscripts start at *zero*, so that the valid following are valid elements of the above declared array are

```
my_array[0], my_array[1], ..., my_array[49]
velocity[0], velocity[1], ..., velocity[99]
```

Note also that C makes no attempt to verify the validity of the indexed element in an array so the burden is all on the programmer to make sure his or her code does not access an invalid memory address.

The following example populates an array of integer then prints its elements using `for` loops:

```
/* Program w5-1.c */

/* This is a simple example with an integer array */

#include <stdio.h>

main() {

  /* declare array */
  int array[10], i;

  /* populate array */
  for (i = 0; i < 10; i++)
    array[i] = 1 + i * i;

  /* print array */
  for (i = 0; i < 10;)
    printf("%d ", array[i++]);

  /* print 6th element */
  printf("\nsixth element is %d\n", array[5]);

}
```

The output is as follows

```
1 2 5 10 17 26 37 50 65 82
sixth element is 26
```

In order to initialize arrays we use the syntax:

```
  type array_name[size] = { value_list };
```

Here are some examples:

```
  int i[5] = {1, 4, 9, 16, 30};
  char a[3] = {'A', 'B', 'C'};
  double d[] = {1., -3., 3.1415};
```

Note that the last initialization is performed without declaring the size of the array. The compiler will make sure enough space is allocated for the array, in this case, 3 elements. The initialization list may be smaller but not larger than the declared array size, if one is present in the declaration.

Arrays are typically used in programs as *fixed-size lists* where one stores data. Entities typically found in scientific programing are also represented as arrays,

such as vectors and matrices. The following program computes the distance, the inner product and angle between two three dimensional vectors:

```c
/* Program w5-2.c */

/* This programs computes the distance, the inner product and angle
   between two 3 dimensional vectors */

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

main() {

  /* declare variables */
  int i;
  double x[3] = {1., 3., -2.};
  double y[3] = {5., 0, .2};
  double tmp;

  double norm_x, norm_y, distance, inner_product, angle;

  /* compute distance and inner product */
  distance = inner_product = norm_x = norm_x = 0.;
  for (i = 0; i < 3; i++) {

    /* compute difference */
    tmp = x[i] - y[i];

    /* compute squared norm of the difference */
    distance += tmp * tmp;

    /* compute squared norm of x and y */
    norm_x += x[i] * x[i];
    norm_y += y[i] * y[i];

    /* compute inner product <x, y> */
    inner_product += x[i] * y[i];

  }

  /* compute square roots of the norms */
  norm_x = sqrt(norm_x);
  norm_y = sqrt(norm_y);
  distance = sqrt(distance);

  /* compute angle */
  angle = acos(inner_product / (norm_x * norm_y));
```

```
  /* print distance, inner product and angle */
  printf("Distance      = %2f\n", distance);
  printf("Inner product = %2f\n", inner_product);
  printf("Angle         = %2f degrees\n", angle * 180 / M_PI);

}
```

Recall that the Euclidean norm, the standard inner product and angle between two 3-dimensional vectors is given by the formulas:

$$\|x\| = \sqrt{\sum_{i=1}^{3} x_i^2}, \qquad \langle x, y\rangle = \sum_{i=1}^{3} x_i y_i, \qquad \angle x, y = \arccos \frac{\langle x, y\rangle}{\|x\|\|y\|}.$$

5.2 Pointers

A pointer is a variable that holds the memory address of another object. If a variable called p contains the address of another variable called q, then p is said to *point to* q. As we will see in this section, understanding pointers is fundamental to understanding arrays in C.

In order to declare a pointer variable precede it by the * operator as in

```
  type *variable_name;
```

For example:

```
  int *p;
  double *q1, *q2;
  void *v;
```

declare a pointer to an integer variable (p), two pointers to double variables (q1 and q2) and one pointer without a specific type (v).

When declared uninitialized, pointers do not hold valid memory addresses. In order to assign the address of one existing variable to a pointer, use the operator *address-of* (&). Recall that & returns the address of the variable it precedes. This can be done at the pointer declaration or anywhere in your program through assignment. For example:

```
  int q;
  int *p1;        /* p1 is a pointer not initialized */
  int *p2 = &q;   /* p2 is a pointer holding the address of q */

  p1 = &q;        /* now p1 also holds the address of q */
```

In order to access the memory address held by a pointer we use the *dereference* or *indirection* operator (*). The operator * returns the *value* stored at the pointer it precedes as well as serves as a proxy for the contents of that memory address during assignment operations. This is illustrated in the next simple program:

```c
/* Program w8-6.c */

#include <stdio.h>

main() {

  /* declare variable and pointer */
  double q, r, *p = &q;

  /* assign value to variables */
  q = 1.34;
  r = 6.78;

  /* print variable, pointer and contents of pointer */
  printf("  value of q = %f\n", q);
  printf("address of q = %p\n", &q);
  printf("  value of r = %f\n", r);
  printf("address of r = %p\n", &r);
  printf("  value of p = %p\n", p);
  printf("address of p = %p\n", &p);
  printf(" value of *p = %f\n", *p);

  /* change variable value */
  q = -7.89;

  /* print variable, pointer and contents of pointer */
  printf("\n  value of q = %f\n", q);
  printf("address of q = %p\n", &q);
  printf("  value of r = %f\n", r);
  printf("address of r = %p\n", &r);
  printf("  value of p = %p\n", p);
  printf("address of p = %p\n", &p);
  printf(" value of *p = %f\n", *p);

  /* change variable value through pointer */
  *p = 12.86;

  /* print variable, pointer and contents of pointer */
  printf("\n  value of q = %f\n", q);
  printf("address of q = %p\n", &q);
  printf("  value of r = %f\n", r);
```

```
  printf("address of r = %p\n", &r);
  printf("  value of p = %p\n", p);
  printf("address of p = %p\n", &p);
  printf(" value of *p = %f\n", *p);

  /* change value of pointer and variable r */
  p = &r;
  r = 5.69;

  /* print variable, pointer and contents of pointer */
  printf("\n  value of q = %f\n", q);
  printf("address of q = %p\n", &q);
  printf("  value of r = %f\n", r);
  printf("address of r = %p\n", &r);
  printf("  value of p = %p\n", p);
  printf("address of p = %p\n", &p);
  printf(" value of *p = %f\n", *p);

}
```

A common mistake is to dereference a pointer when assigning it a value. The following two constructions are equivalent:

```
  int *p;
  p = &q;
```

and

```
  int *p = &q;
```

whereas

```
  int *p;
  p = &q;
```

has a different meaning, usually incorrect.

The arithmetic operators +, ++, -, and -- may be applied to pointer variables. Since pointer variables contain memory addresses you may add or subtract only integer quantities. For example,

```
  double *x;
  ...
  x += 2;
  ...
```

Note that the precedence and binding of * is higher than all arithmetic operators but not higher than ++, hence *p++ increments p not *p, whereas *p+3 adds 3 to *p.

When doing pointer arithmetic, it is sometimes important to know the size of each variable type in bytes. This can vary with the hardware so C provide the keyword `sizeof` that returns the size in bytes of the type or variable that follows it in your current hardware. For example:

```
sizeof(int);
sizeof(float);
sizeof(double);
```

One often needs to use `sizeof` when using `void` pointers. At the end of this excerpt both pointers contain the exact same address:

```
double l[10];
double *x = &l;
void *v = (void *) &l;

x += 2;
v += 2 * sizeof(double);
```

In particular, they both point to the second element of the list `l`. Note the type cast from `(double *)` to `(void *)`.

5.3 Relation between pointers and arrays

In C, pointers and arrays are closely related and often interchangeable. When an array name is used without an index, a pointer to the start of the array is generated. Any operation that can be achieved by array indexing can also be done with pointers. The pointer version is ofter harder to read though.

For example, the declaration:

```
int a[10];
```

defines an array a of size 10. The following statements are equivalent:

```
int *p1, *p2;

p1 = a;
p2 = &a[0];
```

in that both `p1` and `p2` point to the first element of the array `a`. Indeed, `a` itself is a pointer and we can use equivalently pointer arithmetic or array indexing in order to access an element of an array. For example:

```
int *p1, *p2;

p1 = a + 3;
p2 = &a[2];
```

will have p1 pointing to the fourth element of a and p2 pointing to the third element of a. We can also assign values to the array elements by dereferencing the pointer a as in:

```
*a = 2;
*(a + 1) = *a;
*(a + 2) = 4;
```

which will set the first element of a to 2, then copy this value to the second element in the second statement. Finally we set the third element of the array to 4.
Here is an example where an integer array is manipulated through pointers:

```
/* Program w5-1a.c */

/* This is an example with an integer array using pointer
   arithmetic */

#include <stdio.h>

main() {

  /* declare array */
  int array[10], i;

  /* declare pointer */
  int *p;

  /* populate array */
  for (p = array, i = 0; i < 10; i++)
    *(p + i) = i + 1;

  /* print array */
  for (i = 0; i < 10; )
    printf("%d ", *(array + i++));
  printf("\n");

  /* populate array */
  for (i = 10, p = array; p < array + 10; *p++ = i--);

  /* print array */
  for (p = array; p < array + 10; )
    printf("%d ", *p++);
  printf("\n");

}
```

Note the subtleties of incrementing pointers while dereferencing.

5.4 Character arrays and C-strings

C does not have a native *character string* type. Instead, all string manipulation is done through one-dimensional character arrays. The main difference is that most string functions will accept a character array as a string without needing to know its length. Instead, the end of a character string is signaled by the presence of a special character, the *end-of-string character* ('\0') which is simply a character with the decimal value 0. Such character arrays are called *null-terminated strings*. Note that one still has to allocate enough memory space when declaring character arrays regardless of where the end-of-string character is located. For example:

```
char str1[80];              /* Reserves 80 characters for str1 */
char str2[80] = { 'H', 'e', 'l', 'l', 'o', ' ',
                  'w', 'o', 'r', 'l', 'd', '!' };
                            /* Reserves 80 characters for str2
                               and initializes its contents;
                               Does not add end-of-string */
char str3[] = { 'H', 'e', 'l', 'l', 'o', ' ',
                  'w', 'o', 'r', 'l', 'd', '!' };
                            /* Reserves 12 characters for str3
                               and initializes its contents;
                               Does not add end-of-string */
char str4[80] = "Hello world!";
                            /* Reserves 80 characters for str4
                               and initializes its contents;
                               Adds end-of-string */
char str5[] = "Hello world!";
                            /* Reserves 13 characters for str4
                               and initializes its contents;
                               Adds end-of-string */
```

Note that `str5` is a 13 character array rather than a 12 character array because one extra character is allocated and initialized as the end of string character. This is a feature of `str2` being initialized as a *character string* using double quotes ("). In contrast, `str2` and `str3` are initialized as *character arrays*, and therefore their constructor behaves as a standard array constructor. Note also that `str2` and `str3` will not work properly on most C-string functions that *expect* an end-of-string character.

C provides several functions for reading character strings from the terminal, such as `gets` and `scanf` with the '%s' modifier. **THEY ARE ALL UNSAFE AND SHOULD NOT BE USED!** The reason why they are unsafe is because

they do not let the user specify the maximum number of characters in the user character array and this can and have been extensively exploited by malicious software. One should only use functions that allow explicit control of the maximum size of the input to the read. One such function is `fgets`. The following is an example:

```
/* Program w5-5.c */

/* Uses a character array to process a string. */

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

main() {

  char str[81];
  int size;

  printf("Enter a string:\n");
  fgets(str, 81, stdin);

  printf("You entered: '%s'\n", str);

  size = strlen(str);
  printf("Length of string is %d\n", size);

}
```

The meaning of `stdin` in `fgets` will be clear when we study files. A couple of details on using `fgets`: the newline character '`\n`' is part of the string; the function `fgets` returns a pointer to the string read; if none than it returns the null pointer '`NULL`'. This can be used to terminate terminal input. Note that we added the header `<string.h>` so as to access the library function `strlen` to compute the length of the string; this is part of the standard C library so there is no need to add any `-l` flag at compilation. Note also the use of the modifier '`%s`' with `printf` which is used to print a null-terminated string.

The next improved version of the previous code takes care of those issues:

```
/* Program w5-5a.c */

/* Uses a character array to process a string (better version). */

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

```
main() {

  const int BUFFER_SIZE = 80 + 1;

  char str[BUFFER_SIZE];
  int size;

  printf("Enter a string:\n");

  /* loop until useful string is input */
  while ( fgets(str, BUFFER_SIZE, stdin) ) {

    size = strlen(str);

    /* remove \n */
    if ( str[size - 1] == '\n' )
      /* terminate string one character ahead */
      str[size-- - 1] = '\0';

    if ( size > 0 ) {
      printf("You entered: '%s'\n", str);
      printf("Length of string is %d\n", size);
      break;
    } else
      printf("Enter a string:\n");

  }
}
```

After one hits <ENTER> we first remove the trailing '\n' and decrement the size of the string. If the length of `str` is 0, the loop will keep prompting you to enter a non-empty string.

5.5 Multi-dimensional arrays

C allows the definition of multidimensional arrays using the following syntax:

```
type variable_name[size1][size2]...[sizem];
```

For example, to create a 10×12 two-dimensional integer array called `count`, i.e. with 10 rows and 12 columns, the declaration is

```
int count[10][12];
```

In science and engineering analysis, we often deal with matrices. Even though matrices can be thought as the quintessential two dimensional array, using a two-dimensional array to represent matrices is often a bad choice and may lead

to poor performance in today's hardware. Virtually all high-performance computation done with matrices adopts a *flat* one-dimensional representation for matrices. We will see some of this later in the course. The reason why two-dimensional arrays can lead to poor performance is the way the elements are stored and retrieved in the memory, one row at a time. Nevertheless, multidimensional arrays are helpful structures in contexts other than matrices, as we will see.

A three-dimensional array is declared as:

```
float values[10][12][8];
```

Multidimensional arrays can be initialized using the syntax:

```
int svar[3][5] = {
  {1, 0, 3, -1, 3},
  {4, 9, 6, 0, 8},
  {7, 4, 9, 1, -10}
};
```

However, the inner braces are mostly cosmetic, as the compiler will perform the exact same initialization out of

```
int svar[3][5] = { 1, 0, 3, -1, 3, 4, 9, 6, 0, 8, 7, 4, 9, 1, -10 };
```

As a convenience, just as in one-dimensional arrays, you may initialize the array by specifying all but the leftmost dimensions. The following excerpt

```
int svar[][5] = { 1, 0, 3, -1, 3, 4, 9, 6, 0, 8, 7, 4, 9, 1, -10 };
```

would declare and initialize the exact same 3×5 two-dimensional array as before. Note that the number of columns is needed to compute the row number.

Elements in a multidimensional array are accessed one dimension at a time, from left to right. In the case of two-dimensional arrays, this means one row at a time. For example:

```
int svar[3][5] = {
  {1, 0, 3, -1, 3},
  {4, 9, 6, 0, 8},
  {7, 4, 9, 1, -10}
};

svar[1][3] = svar[2][0] = 5;
```

will set the elements in the second row and forth column and third row and first column of `svar` to 5. Recall that indexing starts at 0!

The next program illustrate the use of two-dimensional array to store a matrix of doubles:

```
/* Program w5-9.c */

/* Illustrates two-dimensional arrays */

#include <stdio.h>

main() {

  /* declare variables */
  int i, j;

  /* declare two-dimensional array */
  double array[3][4] = { {3, 7, 4, 6},
                         {-1, 2, 8, .5},
                         {0, -3, 2, 10} };

  /* print matrix */
  for (i = 0; i < 3; i++) {

    /* print rows */
    for (j = 0; j < 4; j++)

      /* print element i, j */
      printf("%12.4f ", array[i][j]);

    /* end of row */
    printf("\n");

  }

}
```

5.6 Arrays of pointers

The correspondence between pointers and arrays is a bit more subtle in the case of multidimensional arrays. Consider the example:

```
  int array[3][5];
  int *p1, *p2, *p3;

  p1 = &array[0][0];
  p2 = array[0];
  p3 = *array;
```

All pointers `p1`, `p2` and `p3` point to the first element of array `array`. Note that a "complete" number of brackets will return an array element, which explains `p1`. The other pointer assignments can be understood by noting that the underlying type for a multidimensional array is that of a *pointer-to-pointer*. That is illustrated below in the case of a two-dimensional array:

```
int array[3][5];
int **p = array;
```

Therefore, `array[0]` and `*array` should dereference a pointer-to-pointer, returning in the above case a pointer to an `int`.

The main difference between multidimensional arrays and arrays of pointer is however the fact that multidimensional arrays are *rectangular* data structures, with the same number of elements allocated for each specified dimension. This constraint is not necessarily present in an array of pointers. This is illustrated in the next example, where a *non-rectangular* array is created using an array of pointers:

```
/* Program w5-6.c */

/* Illustrates arrays of pointers to ints */

#include <stdio.h>

main() {

  /* declare variables */
  int i, j, n;

  /* declare rows */
  /* the first element in a row is the number of elements to follow */
  int row1[] = {3, 7, 4, 6};
  int row2[] = {2, -1, 3};
  int row3[] = {5, 1, 2, 3, 4, 5};

  /* declare array of pointers to hold non-rectangular array */
  int *array[] = { row1, row2, row3 };

  for (i = 0; i < 3; i++) {

    /* print row label */
    printf("Row %d:\n", i + 1);

    /* get number of elements in the row */
    n = array[i][0];
```

```
    /* print elements of the row */
    for (j = 1; j <= n; j++)
      printf("%d ", array[i][j]);

    printf("\n");

  }

}
```

The next example uses an array of pointers to hold strings of different lengths:

```
/* Program w5-7.c */

/* Illustrates arrays of pointers to strings */

#include <stdio.h>

main() {

  /* declare variables */
  int i;
  char *messages[] = { "MAE 9 rocks!",
                       "How I like my tutor!",
                       "I was born to program in C." };

  for (i = 0; i < 3; i++)
    printf("Message %d: '%s'\n", i + 1, messages[i]);

}
```

This construction is so commonly used that it is the mechanism by which command line parameters are passed to a C program. The next example illustrates this:

```
/* Program w5-8 */

/* Illustrate the use of command line parameters */

#include <stdio.h>

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

  int i;

  for (i = 0; i < argc; i++)
    printf("Parameter #%d: '%s'\n", i + 1, argv[i]);
```

```
}
```

The above program produces

```
iacs5.ucsd.edu% a.out 9 parameter -k
Parameter #1: 'a.out'
Parameter #2: '9'
Parameter #3: 'parameter'
Parameter #4: '-k
```