

A Rudimentary Intro to C programming

Wayne Goddard

School of Computing, Clemson University, 2008

Part 3: Arrays and Functions

12	Arrays	C1
13	Two Examples	C3
14	Functions	C6
15	More on Functions	C8
16	More on Arrays and Functions	C10
17	2-Dimensional Arrays	C14

Arrays

12.1 Creating an Array

Suppose we wanted to store 100 scores. One might try to declare 100 variables, say `score0 ... score99`. However, it is much better to define an array.

An array is a collection of variables of the same type. These values are accessed by an *index*, also called *subscript*. The declaration

```
float score[100];
```

creates an array with 100 values: these are `score[0]` through `score[99]`. The 0 and the 99 are the index/subscript. Note that a C array always starts at 0.

An array value can be used wherever a normal value is used. Here is code to calculate the sum of the above array:

```
float total = 0.0;
for(int i=0; i<100; i++) {
    total += score[i];
}
```

Like a normal variable, an array needs to be initialized before use. For example, one might write:

```
const int NUM_VALS = 20;
int count[NUM_VALS];
for(int j=0; j<NUM_VALS; j++)
    count[j] = 0;
```

to set all the values in the array to zero.

Occasionally, an array can be initialized at declaration:

```
char legalGrades[6] = { 'A', 'B', 'C', 'D', 'F', 'I' };
```

Note that C does no range checking: you must ensure that the index is valid.

12.2 Sample program: `arrayed.c`

Here is code that reads an array from the user, then prints the array, and then determines whether every entry in the array is the same or not. This introduces the old style of constants—`#define`—and the command `break` to get out of a loop prematurely.

```

// read an array of 10 integers and check if all the same
#include <stdio.h>
#include <stdbool.h>
#define SIZE 10

int main(void) {
    // define and get array
    int A[SIZE];
    printf("Enter %d values\n", SIZE);
    int i;
    for(i=0; i<SIZE; i++)
        scanf("%d", &A[i]);

    // echo array
    printf("You entered: ");
    for(i=0; i<SIZE; i++)
        printf("%d ", A[i]);
    printf("\n");

    // check for sameNess
    bool allAreSame=true;
    for(i=0; i<SIZE-1; i++) { // note where loop stops
        if( A[i]!=A[i+1] ) {
            allAreSame=false;
            break; // terminates loop prematurely
        }
    }
    if( allAreSame )
        printf("Same\n");
    else
        printf( "not-same\n" );

    return 0;
}

```

Practice Change the above program to test whether the input numbers are consecutive or not (rather than whether they are all the same).

Two Examples

13.1 Random Numbers

The function `rand()` from the standard library generates a random nonnegative integer. Often one wants a random number in a particular range; this can be achieved by taking the random number modulo the size of the range.

If you run the program twice, it will use the same sequence of random numbers—these are actually produced by what is called a pseudorandom number generator. You can avoid this repetition by seeding the generator: either by typing in a random number yourself or using the internal clock. Note that the generator should not be seeded more than once.

Here is a program to simulate 100 rolls of a dice and report the statistics.

```
// dice.c - wdg 2008
// rolls a die 100 times and prints the statistics
#include <stdio.h>
#include <stdlib.h> // for random number generator
#include <time.h> // for time function

const int MAX = 100;
const int SIDES = 6;

int main(void) {

    int i,j;
    srand(time(0)); // seed random number generator via internal clock
    int count[SIDES];
    for(i=0; i<SIDES; i++)
        count[i]=0;

    for(j=0; j<MAX; j++) {
        int roll = (rand() % SIDES) + 1;
        printf("%d",roll);
        count[ roll-1 ]++;
    }

    printf("\n");
}
```

```

for(i=0; i<SIDES; i++)
    printf("%d came up %d times\n", i+1, count[i]);

return 0;
}

```

13.2 Generating Prime Numbers

Consider the problem of determining the first 50 prime numbers. Recall that a prime number is a number bigger than 1 whose only divisors are itself and 1.

One way to do this is called the Sieve of Erasthenes (see Wikipedia). But we will instead do this in the naive obvious way: consider each number in turn and see if its prime.

To see if a number is prime, it suffices to check whether it is divisible by any of the prime numbers smaller than it. So we maintain an array of the prime numbers found so far. We also keep track of how many primes already found.

The program is provided below. Note that we do not have to initialize the array `prime`: we only ever access the first `numFound` entries, and these have been filled by us.

```

// prints out first 50 primes using naive inefficient algorithm
// wdg 2008
#include <stdio.h>
#include <stdbool.h>

int main(void) {

    const int MAX=50;
    int prime[MAX], numFound=0, contender, p;
    bool contenderSeemsPrime;

    for( contender=2; numFound<MAX; contender++) { // note continue condn.

        contenderSeemsPrime = true;
        for( p=0; p<numFound && contenderSeemsPrime; p++ ) {
            if( contender % prime[p] == 0 )
                contenderSeemsPrime = false;
        }

        if( contenderSeemsPrime ) {
            printf("%d ", contender);

```

```
        prime[ numFound ] = contender;
        numFound++;
    }

} // end for

printf("\n");
return 0;
}
```

Functions

14.1 Function Code

A *function* is a self-standing piece of code. It optionally takes some parameters and optionally returns a value. The general syntax is:

```
type function_name( parameters ) {
    ... some code ...
    a return statement if needed
}
```

The simplest form of a function just does an independent piece of work without interacting with the rest of the program. This has type `void`. For example:

```
void say_hello(void) {
    printf("Hello");
}
int main(void) {
    say_hello();
    say_hello();
    return 0;
}
```

will print out HelloHello. We say that the `main` function *calls* the `say_hello` function.

14.2 Function Parameters

The next level of function is to do something flexible. For example, a function might print out the sum of the first so many squares. The value passed to the function is called a *parameter* or *argument*. The following program will print out the values 14 and 55.

```
void outputSumOfSquares(int n) {
    int i, sum=0;
    for(i=1; i<=n; i++)
        sum += i*i;
    printf("Sum of squares is %d", sum);
}
```

```

int main(void) {
    outputSumOfSquares(3);
    outputSumOfSquares(5);
    return 0;
}

```

The main function begins with the call `outputSumOfSquares(3)`. This starts the code for that function running with the value 3 for `n`. When the function finishes, execution passes back to the main function. This function then calls `outputSumOfSquares` again; this time that function runs with `n` having the value 5.

Note that the function has its own collection of variables: `n`, `sum`, `i`. When the function starts, the value of `n` is already initialized by the system to a particular value: we do NOT have to read the value from the user.

The `printf` and `scanf` commands are actually functions.

14.3 Sample Program: `gcd.c`

Euclid developed an algorithm to calculate the greatest common divisor of two numbers. The idea is to repeatedly subtract the smaller number from the larger number until one of them becomes zero; at that point the other number is the gcd. We speed up the repeated subtraction by dividing the larger by the smaller and keeping the remainder.

```

/* program for gcd of two numbers using Euclid's algorithm
   based on Kochan - wdg 2008 */
#include <stdio.h>

void calcGCD( int u, int v) {
    int temp;
    while( v!=0 ) {                // think of v as the smaller
        temp = u % v;
        u = v;
        v = temp;
    }
    printf("Greatest common divisor is %d\n", u);
}

int main(void) {
    calcGCD( 3, 5 );
    calcGCD( 22, 44);
    return 0;
}

```

More on Functions

15.1 Returning Values

The functions already discussed do not produce values, but we have seen that the mathematics library has functions that produce values. A function has a *type*: the type may be a standard data type, such as `int` or `float`, or the type `void` which means it does not return anything.

The value is returned by a `return` statement which is usually (but does not have to be) at the end of the function. A `void` function may also have a `return` statement for premature termination, though `returns not at the end` should be used sparingly.

Here is example code for a function which returns the minimum of two values:

```
int minimum ( int v1, int v2 ) {
    if( v1<v2 )
        return v1;
    else
        return v2;
}
```

Note the `v1` and `v2` exist only inside the function: their *scope* is *local*. If one were to change the value of `v1` in the function, it would NOT affect the value of the variable that was passed to it.

15.2 Stubs and Prototypes

Prototypes are used to tell the compiler about the functions that are available. A prototype might be

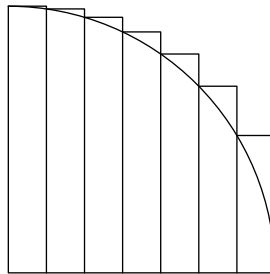
```
int minimum(int,int);
```

(Note the semi-colon!) Actually including the names of the parameters helps the reader. Prototypes are needed if the call to the function occurs before the function code. Some advocate prototypes for all functions. Ideally these should be in a *header* file, discussed in a later chapter.

A *stub* is useful while developing the code: it is a function that compiles but does nothing.

15.3 Sample Program: archimedes.c

Archimedes tried to estimate π . His idea was to consider a quarter of the circle (which has area $\pi/4$). Then he divided the circle into parallel vertical strips and estimated the area as the sum of the strips. If one takes the left-endpoint (as below) one gets an over-estimate. The program also goes awry when the divisions are small, due to numeric problems.



```
// program to replicate Archimedes' calculation of pi
// also shows numerical problems - wdg 2008
#include <stdio.h>
#include <math.h>

float estimate(int); // prototype

int main(void) {

    int divs;
    for(divs=1; divs<2000000; divs*=2) {
        float pi = estimate(divs);
        printf("For division=%d Pi is approximately %.5f\n", divs, pi );
    }
    return 0;
}

float estimate(int divisions) {
    int i;
    float sum =0.0, width = 1.0/divisions;
    for( i=0; i<divisions; i++ ) {
        sum += width * sqrt( 1.0 - (i*width)*(i*width) );
    }
    return 4*sum;
}
```

More on Arrays and Functions

16.1 Arrays as Function Parameters

An array can be the parameter of a function. An array passed as a parameter to a function does not have to have its size declared. Instead it is common to specify the size of the array as a second parameter. For example, here is code to check whether a given array is sorted or not:

```
int isSorted(float A[], int size) {
    int i;
    for(i=0; i<size-1; i++) {
        if(A[i]>A[i+1])
            return 0;    // false
    }
    return 1;           // true
}
```

This function uses the style where boolean variables are simulated by integers. Note that changes to the contents of the array in the function DO affect the original array (why is discussed in a later chapter).

16.2 Sample code: linearSearch.c

A *linear search* is just a fancy name for the idea of looking for something by going through the array until it is found. The following code uses the `sizeof` operator: this is a compile-time operator, and is only allowed if the compiler can determine the array size.

```
/* linear search in an array */
#include <stdio.h>

int indexOf(int A[], int size, int val);

int main(void)
{
    int test[] = {1,3,5,7,2,4,6};
    int testVal, position;
    int arraySize = sizeof(test)/sizeof(int);
```

```

    testVal=3;
    position = indexOf(test, arraySize, testVal) ;
    printf("%d is at %d\n", testVal, position );
    testVal=33;
    position = indexOf(test, arraySize, testVal) ;
    printf("%d is at %d\n", testVal, position );
    return 0;
}

// returns index of first occurrence of val in array A (-1 if failure)
int indexOf(int A[], int size, int val) {
    int i;
    for(i=0; i<size; i++) {
        if( val == A[i] )
            return i;
    }
    // failing which
    return -1;
}

```

16.3 Multiple Functions

In large programs, it is standard to have functions calling functions calling functions etc. For us, if we have multiple functions we will put them all in the same file, and put prototypes for all at the top.

Also each function must have a comment explaining its purpose, parameters, returns, where applicable. Some people like to have pre-conditions and post-conditions: a *pre-condition* is something that is assumed to be true before the function is called, and a *post-condition* is what has changed after the function is called.

16.4 Sample Program: `extremumArray.c`

Here is a simple program that reads an array and prints out the minimum and maximum values in the array.

```

// read array and calculate minimum/maximum
// wdg 2008
#include <stdio.h>
#include <stdbool.h>

```

```

const int MAX = 100;

int readArray( float arr[] );
float extremum( float arr[], int count, bool wantMax );

int main( void ) {
    float A[MAX];
    int len = readArray(A);
    printf("Minimum is %f\n", extremum(A,len,false) );
    printf("Maximum is %f\n", extremum(A,len,true) );
    return 0;
}

// reads an array of floats and returns the size of the array
int readArray( float arr[] ) {
    int i, length;

    printf("Enter array size ");
    scanf("%d", &length);
    while( length < 0 || length > MAX ) {
        printf("Size should be in range 1 to %d. Retry: ", MAX);
        scanf("%d", &length);
    }

    printf("Enter values ");
    for(i=0; i<length; i++)
        scanf("%f", &arr[i] );

    return length;
}

// calculates minimum or maximum of array arr depending on wantMax
// assumes arr has count>0 elements
float extremum( float arr[], int count, bool wantMax ) {
    float bestSoFar = arr[0];
    int i;
    for(i=1; i<count; i++) {
        if( (wantMax && arr[i]>bestSoFar) ||
            (!wantMax && arr[i]<bestSoFar) )

```

```
        bestSoFar = arr[i];  
    }  
    return bestSoFar;  
}
```

2-Dimensional Arrays

17.1 2-Dimensional Arrays

A two-dimensional array can be used to represent, for example, a matrix, a table or a chess board. The row and column positions are given as successive indices. For example, a board for tictactoe might have something like

```
char board[3][3];
board[1][1] = 'X'; // the center
```

If two-dimensional arrays are passed as parameters, at least their first dimension size needs to be specified; for our examples, we will specify both sizes.

17.2 Sample code: chess.c

A rook in chess attacks all squares in the same row and in the same column. The following code shows where some placed rooks can attack.

```
// chess.c - wdg
// program to show a little 2D array stuff
#include <stdio.h>

void initBoard( char B[8][8] );
void placeRook( char B[8][8], int row, int col );
void printBoard( char B[8][8] );

const char CLEAR = '.';
const char ROOK = 'R';
const char ATTACKED = 'x';

int main(void) {
    char board[8][8];
    initBoard(board);
    placeRook(board,3,4);
    placeRook(board,5,0);
    printBoard(board);
    return 0;
}
```

```

// @pre: none
// @post: all entries in B are set to a period
void initBoard( char B[8][8] ) {
    int i,j;
    for(i=0; i<8; i++)
        for(j=0; j<8; j++)
            B[i][j] = CLEAR;
}

// @pre: none
// @parms: assumes 0<= row, col < 8
// @post: rook added to board B at square indicated
void placeRook( char B[8][8], int row, int col ) {
    int i;
    for(i=0; i<8; i++)
        B[row][i] = B[i][col] = ATTACKED;
    B[row][col] = ROOK;
}

// @pre: none
// @post: contents of B printed on standard output
void printBoard( char B[8][8] ) {
    int i,j;
    for(i=0; i<8; i++) {
        for(j=0; j<8; j++)
            printf("%c", B[i][j]);
        printf("\n");
    }
}

```

Practice. The `placeRook` function has a bug (think about rooks already on the board). It also does not validate its parameters. Revise the function to improve it.