Arrays

1

Using the `sizeof` Operator with Arrays

- The `sizeof` operator can determine the size of an array (in bytes).
- If `a` is an array of 10 integers, then `sizeof(a)` is typically 40 (assuming that each integer requires four bytes).
- We can also use `sizeof` to measure the size of an array element, such as `a[0]`.
- Dividing the array size by the element size gives the length of the array:

```
sizeof(a) / sizeof(a[0])
```

2

Using the `sizeof` Operator with Arrays

- Some programmers use this expression when the length of the array is needed.
- A loop that clears the array `a`:

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
  a[i] = 0;
```

  Note that the loop doesn't have to be modified if the array length should change at a later date.

3

Using the `sizeof` Operator with Arrays

- Some compilers produce a warning message for the expression `i < sizeof(a) / sizeof(a[0])`.
- The variable `i` probably has type `int` (a signed type), whereas `sizeof` produces a value of type `size_t` (an unsigned type).
- Comparing a signed integer with an unsigned integer can be dangerous, but in this case it's safe…
- How to avoid this?

4

Using the **`sizeof`** Operator with Arrays

- To avoid a warning, we can add a cast that converts `sizeof(a) / sizeof(a[0])` to a signed integer:

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
  a[i] = 0;
```

- Defining a macro for the size calculation can be helpful:

```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

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

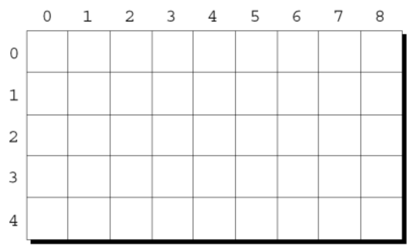
5

Multidimensional Arrays

- An array may have any number of dimensions.
- The following declaration creates a two-dimensional array (a *matrix*, in mathematical terminology):

```
int m[5][9];
```

- `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:



6

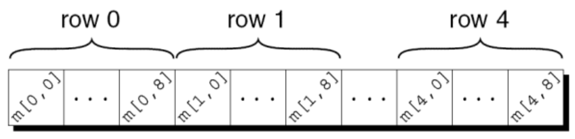
Multidimensional Arrays

- To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`.
- The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.
- Resist the temptation to write `m[i,j]` instead of `m[i][j]`.

7

Multidimensional Arrays

- Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- How the `m` array is stored:



8

Multidimensional Arrays

- Nested `for` loops are ideal for processing multidimensional arrays.
- Consider the problem of initializing an array for use as an identity matrix. A pair of nested `for` loops is perfect:

```
#define N 10 /* vs 'const int N = 10;'*/

int ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++){
    if (row == col)
      ident[row][col] = 1;
    else
      ident[row][col] = 0;
   }
```

Can this be made more efficient?

```
ident[N][N] = {0};
for (int i=0;i<N;i++)
ident[i][i] = 1;
```

9

Initializing a Multidimensional Array

- We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- Initializers for higher-dimensional arrays are constructed in a similar fashion.
- C provides a variety of ways to abbreviate initializers for multidimensional arrays

10

Initializing a Multidimensional Array

- If an initializer isn’t large enough to fill a multidimensional array, the remaining elements are given the value 0.
- The following initializer fills only the first three rows of `m`; the last two rows will contain zeros:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

11

Initializing a Multidimensional Array

- If an inner list isn’t long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1},
               {0, 1, 0, 1, 1, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

12

Initializing a Multidimensional Array

- We can even omit the inner braces:

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

  Once the compiler has seen enough values to fill one row, it begins filling the next.
- Omitting the inner braces can be risky, since an extra element (or even worse, a missing element) will affect the rest of the initializer.

13

Initializing a Multidimensional Array

- C99's designated initializers work with multidimensional arrays.
- How to create 2 × 2 identity matrix:

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

  As usual, all elements for which no value is specified will default to zero.

14

Constant Arrays

- An array can be made "constant" by starting its declaration with the word `const`:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

- An array that's been declared `const` should not be modified by the program.
- `const` isn't limited to arrays, but it's particularly useful in array declarations.

15

Program: Dealing a Hand of Cards

- The `deal.c` program illustrates both two-dimensional arrays and constant arrays.
- The program deals a random hand from a standard deck of playing cards.
- Each card in a standard deck has a *suit* (clubs, diamonds, hearts, or spades) and a *rank* (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace).

16

Program: Dealing a Hand of Cards

- The user will specify how many cards should be in the hand:

```
Enter number of cards in hand: 5
Your hand: 7c 2s 5d as 2h
```

- Problems to be solved:
  - How do we pick cards randomly from the deck?
  - How do we avoid picking the same card twice?

17

Program: Dealing a Hand of Cards

- To pick cards randomly, we'll use several C library functions:
  - `time` (from `<time.h>`) – returns the current time, encoded in a single number.
  - `srand` (from `<stdlib.h>`) – initializes C's random number generator.
  - `rand` (from `<stdlib.h>`) – produces an apparently random number each time it's called.
- By using the `%` operator, we can scale the return value from `rand` so that it falls between 0 and 3 (for suits) or between 0 and 12 (for ranks).

18

Program: Dealing a Hand of Cards

- The `in_hand` array is used to keep track of which cards have already been chosen.
- The array has 4 rows and 13 columns; each element corresponds to one of the 52 cards in the deck.
- All elements of the array will be false to start with.
- Each time we pick a card at random, we' ll check whether the element of `in_hand` corresponding to that card is true or false.
  - If it' s true, we' ll have to pick another card.
  - If it' s false, we' ll store `true` in that element to remind us later that this card has already been picked.

19

Program: Dealing a Hand of Cards

- Once we've verified that a card is "new", we'll need to translate its numerical rank and suit into characters and then display the card.
- To translate the rank and suit to character form, we'll set up two arrays of characters—one for the rank and one for the suit—and then use the numbers to subscript the arrays.
- These arrays won't change during program execution, so they are declared to be `const`.

20

deal.c

```
/* Deals a random hand of cards */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

const int NUM_SUITS 4;
const int NUM_RANKS 13;

int main(void)
{
  bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
  int num_cards, rank, suit;
  const char rank_code[] = {'2','3','4','5','6','7','8',
                            '9','t','j','q','k','a'};
  const char suit_code[] = {'c','d','h','s'};
```

```
  srand((unsigned) time(NULL));

  printf("Enter number of cards in hand: ");
  scanf("%d", &num_cards);

  printf("Your hand:");
  while (num_cards > 0) {
    suit = rand() % NUM_SUITS;    /* picks a random suit */
    rank = rand() % NUM_RANKS;    /* picks a random rank */
    if (!in_hand[suit][rank]) {
      in_hand[suit][rank] = true;
      num_cards--;
      printf(" %c%c", rank_code[rank], suit_code[suit]);
    }
  }
  printf("\n");

  return 0;
}
```

Functions

23

Introduction

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.
- Advantages of functions:
  - A program can be divided into small pieces that are easier to understand and modify.
  - We can avoid duplicating code that's used more than once.
  - A function that was originally part of one program can be reused in other programs.

24

Program: Computing Averages

- The `average.c` program reads three numbers and uses the `average` function to compute their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

25

average.c

```
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

26

Program: Printing a Countdown

- To indicate that a function **has no return value**, we specify that its return type is **`void`**:

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

- `void` is a type with no values.
- A call of `print_count` must appear in a statement by itself:

```
print_count(i);
```

- The `countdown.c` program calls `print_count` 10 times inside a loop.

27

`countdown.c`

```
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```

28

Program: Printing a Pun

- When a function has **no parameters**, the word **`void`** is placed in parentheses after the function's name:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}
```

- To call a function with no arguments, we write the function's name, followed by parentheses:

```
print_pun();
```

The parentheses *must* be present.

- The `pun2.c` program tests the `print_pun` function.

29

pun2.c

```
/* Prints a bad pun */

#include <stdio.h>

void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
  print_pun();
  return 0;
}
```

30

Function Definitions

- General form of a ***function definition:***

  return-type *function-name* (*parameters*)
  {
      declarations
      statements
  }

31

Function Definitions

- The **return type** of a function is the type of value that the function returns.
- Rules governing the return type:
  - Functions may **not return arrays**.
  - Specifying that the return type is **`void`** indicates that the function doesn' t return a value.
- If the return type is omitted in C89, the function is presumed to return a value of type `int`.
- In C99, omitting the return type is illegal.

32

Function Definitions

- As a matter of style, some programmers put the return type *above* the function name:

```
double
average(double a, double b)
{
  return (a + b) / 2;
}
```

- Putting the return type on a separate line is especially useful if the return type is lengthy, like `unsigned long int`.

33

Function Definitions

- After the function name comes **a list of parameters**.
- Each parameter is preceded by a specification of its type; parameters are separated by commas.
- If the function has no parameters, the word `void` should appear between the parentheses.

34

Function Definitions

- **Variables declared in the body** of a function can't be examined or modified by other functions.
- In C89, variable declarations must come first, before all statements in the body of a function.
- In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

35

Function Definitions

- The body of a function whose return type is `void` (a "`void` function") can be empty:

```
void print_pun(void)
{
}
```

- Leaving the body empty may make sense as a temporary step during program development.

36

Function Calls

- The value returned by a non-`void` function can always be discarded if it’s not needed:

```
average(x, y);  /* discards return value */
```

This call is an example of an expression statement: a statement that evaluates an expression but then discards the result.

37

Function Calls

- Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense.
- **`printf` returns the number of characters** that it prints.
- After the following call, `num_chars` will have the value 9:

```
num_chars = printf("Hi, Mom!\n");
```

- We’ll normally discard `printf`’s return value:

```
printf("Hi, Mom!\n");
  /* discards return value */
```

38

Function Calls

- To make it clear that we're deliberately discarding the return value of a function, C allows us to put `(void)` before the call:

  ```
  (void) printf("Hi, Mom!\n");
  ```

- Using `(void)` makes it clear to others that you deliberately discarded the return value, not just forgot that there was one.

39

Function Declarations

- Suppose we putt the definition of `average` *after* `main`.

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

40

Function Declarations

- When the compiler encounters the first call of `average` in `main`, it has no information about the function.
- Instead of producing an error message, the compiler assumes that `average` returns an `int` value.
- We say that the compiler has created an ***implicit declaration*** of the function.
- When it encounters the definition of `average` later, the compiler notices the function's return type is `double`, not `int`, and so we get **an error message**.

41

Function Declarations

- One way to avoid the problem of call-before-definition is to arrange the program so that the **definition of each function precedes all its calls.**
- Unfortunately, such an arrangement doesn' t always exist.
- Even when it does, it may make the program harder to understand by putting its function definitions in an unnatural order.

42

Function Declarations

- Fortunately, C offers a better solution: **declare each function before calling it**.
- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

  return-type function-name `(` *parameters* `)` `;`
- Here's the `average.c` program with a declaration of `average` added.

43

Function Declarations

```
#include <stdio.h>

double average(double a, double b);   /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)    /* DEFINITION */
{
  return (a + b) / 2;
}
```

44