Chapter 8

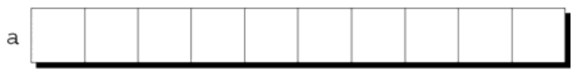
Arrays

Scalar Variables versus Aggregate Variables

- So far, the only variables we've seen are ***scalar:*** capable of holding a single data item.
- C also supports ***aggregate*** (聚合) variables, which can store collections of values.
- There are two kinds of aggregates in C:
  - arrays and
  - structures.
- The focus of the chapter is on
  - one-dimensional arrays,
  - which play a much bigger role in C than do multidimensional arrays.

One-Dimensional Arrays

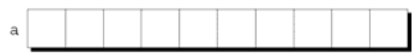
- An ***array*** is a data structure containing a number of data values, all of which have the same type.
- These values, known as ***elements,*** can be individually selected by their position within the array.
- The simplest kind of array has just one dimension.
- The elements of a one-dimensional array `a`
  - are conceptually arranged one after another
  - in a single row (or column):



One-Dimensional Arrays

- To declare an array, we must specify
  - the *type* of the array's elements and
  - the *number* of elements:

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



- The elements may be of any type; the length of the array can be any (integer) constant expression.
- Using a macro to define the length of an array is an excellent practice:

```
#define N 10
…
int a[N];
```

Array Subscripting

```
int a[10], b;

a[9] = 9;
b    = a[9];
```

- To access an array element, write the array name followed by an integer value in square brackets.
- This is referred to as ***subscripting*** or ***indexing*** the array.
- The elements of an array of length *n* are indexed from 0 to *n* – 1.
- If `a` is an array of length 10, its elements are designated by `a[0]`, `a[1]`, …, `a[9]`:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Array Subscripting

- Expressions of the form `a[i]` are lvalues,
  - so they can be used in the same way as ordinary variables:

1									
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

- In general, if an array contains elements of type *T*, then each element of the array is treated as if it were a variable of type *T*.

Array Subscripting

- Many programs contain `for` loops whose job is to perform some operation on every element in an array.
- Examples of typical operations on an array `a` of length `N`:

0	0	0	0	0	0	0	0	0	0
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

```
for (i = 0; i < N; i++)
  a[i] = 0;              /* clears a */

for (i = 0; i < N; i++)
  scanf("%d", &a[i]);    /* reads data into a */

for (i = 0; i < N; i++)
  sum += a[i];           /* sums the elements of a */
```


Array Subscripting

- C doesn’t require that subscript bounds be checked;
  - if a subscript goes out of range, the program’s behavior is undefined.
- A common mistake: forgetting that an array with *n* elements is indexed from 0 to *n* – 1, not 1 to *n*:

```
int a[10], i;

for (i = 1; i <= 10; i++)     /* WRONG */
  a[i] = 0;
```

With some compilers, this innocent-looking (看似無辜的) `for` statement causes an infinite loop.

Array Subscripting

- An array subscript may be any integer expression:

```
a[i + j*10] = 0;
```

- The expression can even have side effects:

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


Array Subscripting

- Be careful when an array subscript has a side effect:

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

- The expression `a[i] = b[i++]`
  - accesses the value of `i`
  - and also modifies `i`,
  - causing undefined behavior.
- The problem can be avoided by removing the increment from the subscript:

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


Program: Reversing a Series of Numbers

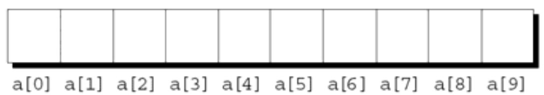
- The `reverse.c` program
  - prompts the user to enter a series of numbers,
  - then writes the numbers in reverse order:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

- The program
  - stores the numbers in an array as they're read,
  - then goes through the array backwards,
  - printing the elements one by one.

`reverse.c`

```
/* Reverses a series of numbers */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = N - 1; i >= 0; i--)
    printf(" %d", a[i]);

  printf("\n");

  return 0;
}
```



Array Initialization

- An array,
  - like any other variable,
  - can be given an initial value at the time it's declared.
- The most common form of ***array initializer***
  - is a list of constant expressions
  - enclosed in braces and separated by commas:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

1	2	3	4	5	6	7	8	9	10
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Array Initialization

- If the initializer is shorter than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1,2,3,4,5,6,0,0,0,0} */
```

- Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
/* initial value of a is {0,0,0,0,0,0,0,0,0,0} */
```

  There's a single 0 inside the braces because it's illegal for an initializer to be completely empty.
- It's also illegal for an initializer
  - to be longer than the array it initializes.

Array Initialization

- If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- The compiler uses the length of the initializer to determine how long the array is.

Designated Initializers (C99)

- It's often the case that
  - relatively few elements of an array need to be initialized explicitly; the other elements can be given default values.
- An example:

```
int a[15] =
  {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

- For a large array, writing an initializer in this fashion is tedious 沈悶費工 and error-prone 易於犯錯.

Designated Initializers (C99)

- C99's ***designated*** (標明) ***initializers*** (指定碼)
  - can be used to solve this problem.
- Here's how we could redo the previous example using a designated initializer:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

- Each number in brackets 中括弧 is said to be a ***designator*** (指定碼).
- [ ] brackets　　( ) parentheses, brackets (UK)
- { } braces　　⟨ ⟩ angle brackets

Designated Initializers (C99)

- Designated initializers
  - are shorter
  - and easier to read (at least for some arrays).
- Also, the order in which the elements are listed no longer matters.
- Another way to write the previous example:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```


Designated Initializers (C99)

- Designators
  - must be integer constant expressions.
- If the array being initialized has length *n*,
  - each designator must be between 0 and *n* – 1.
- If the length of the array is omitted,
  - a designator can be any nonnegative integer.
  - The compiler will deduce (推斷) the length of the array from the largest designator.
- The following array will have 24 elements:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```


Designated Initializers (C99)

- An initializer may use
  - both the older (element-by-element) technique
  - and the newer (designated) technique:

```
int c[10] = {5, 1, 9, [4] = 3, 7, 2, [8] = 6};

// not supported in Dev-C++,
// compilation error!
```

Program: Checking a Number for Repeated Digits

- The `repdigit.c` program
  - checks whether any of the digits in a number appear more than once.
- After the user enters a number, the program prints
  - either `Repeated digit` or `No repeated digit`:

```
Enter a number: 28212
Repeated digit
```

- The number 28212
  - has a repeated digit (2); a number like 9357 doesn't.


```
bool digit_seen[10] = {false};
```

Program: Checking a Number for Repeated Digits

- The program uses an array of 10 Boolean values
  - to keep track of which digits appear in a number.
  - Initially, every element of the `digit_seen` array is false.
- When given a number `n`,
  - the program examines `n`'s digits one at a time,
  - storing the current digit in a variable named `digit`.

28212

- If `digit_seen[digit]` is true,
  - then `digit` appears at least twice in `n`.

F	F	F	F	F	F	F	F	F	F
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

- If `digit_seen[digit]` is false,
  - then `digit` has not been seen before,
  - so the program sets `digit_seen[digit]` to `true`
  - and keeps going.

repdigit.c

```
/* Checks numbers for repeated digits */

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

int main(void)
{
  bool digit_seen[10] = {false};
  int digit;
  long n;

  printf("Enter a number: ");
  scanf("%ld", &n);

  while (n > 0) {
    digit = n % 10;
    if (digit_seen[digit])
      break;
    digit_seen[digit] = true;
    n /= 10;
  }
```

```
28212 % 10 = 2
28212 / 10 = 2821
```

F	F	T	F	F	F	F	F	F	F
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]


```
  if (n > 0)
    printf("Repeated digit\n");
  else
    printf("No repeated digit\n");

  return 0;
}
```

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])
```


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.

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.

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 is often helpful:

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

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

Program: Computing Interest

- The `interest.c` program
  - prints a table showing the value of $100 invested at different rates of interest over a period of years.
- The user will enter
  - an interest rate and
  - the number of years the money will be invested.
- The table will show the value of the money
  - at one-year intervals—at that interest rate and the next four higher rates—
  - assuming that interest is compounded once a year (年複利).

Program: Computing Interest

- Here's what a session with the program will look like:

```
Enter interest rate: 6
Enter number of years: 5

Years     6%     7%     8%     9%    10%
  1     106.00 107.00 108.00 109.00 110.00
  2     112.36 114.49 116.64 118.81 121.00
  3     119.10 122.50 125.97 129.50 133.10
  4     126.25 131.08 136.05 141.16 146.41
  5     133.82 140.26 146.93 153.86 161.05
```


Program: Computing Interest

- The numbers in the second row depend on the numbers in the first row, so it makes sense to store the first row in an array.
  - The values in the array are then used to compute the second row.
  - This process can be repeated for the third and later rows.
- The program uses nested `for` statements.
  - The outer loop counts from 1 to the number of years requested by the user.
  - The inner loop increments the interest rate from its lowest value to its highest value.

interest.c

```
/* Prints a table of compound interest */

#include <stdio.h>

#define NUM_RATES ((int) (sizeof(value) / sizeof(value[0])))
#define INITIAL_BALANCE 100.00

int main(void)
{
  int i, low_rate, num_years, year;
  double value[5];

  printf("Enter interest rate: ");
  scanf("%d", &low_rate);

  printf("Enter number of years: ");
  scanf("%d", &num_years);
```

```
printf("\nYears");
for (i = 0; i < NUM_RATES; i++) {
  printf("%6d%%", low_rate + i);
  value[i] = INITIAL_BALANCE;
}
printf("\n");

for (year = 1; year <= num_years; year++) {
  printf("%3d    ", year);
  for (i = 0; i < NUM_RATES; i++) {
    value[i] += (low_rate + i) / 100.0 * value[i];
    printf("%7.2f", value[i]);
  }
  printf("\n");
}

return 0;
}
```

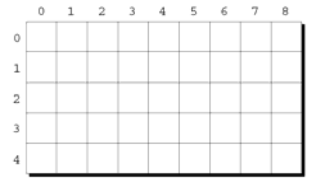
Years	6%	7%	8%	9%	10%
1	106.00	107.00	108.00	109.00	110.00
2	112.36	114.49	116.64	118.81	121.00
3	119.10	122.50	125.97	129.50	133.10
4	126.25	131.08	136.05	141.16	146.41
5	133.82	140.26	146.93	153.86	161.05

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:

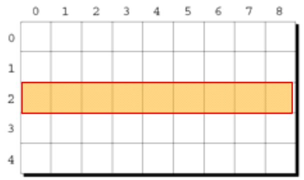


Multidimensional Arrays

- To access the element of `m` in row `i`, column `j`, we must write

```
m[i][j]
```

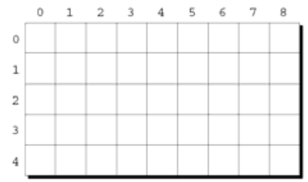
m[i]



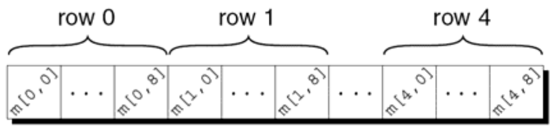
- The expression
  - `m[i]` designates row `i` of `m`, and
  - `m[i][j]` then selects element `j` in this row.
- Resist the temptation to write `m[i,j]  /*誤*/` instead of `m[i][j]`.
  - C treats the comma as an operator in this context, so `m[i,j]` is the same as `m[j]`.

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:



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

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

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

	0	1	2	3	4	5	6	7	8
0	1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0

Initializing a Multidimensional Array

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

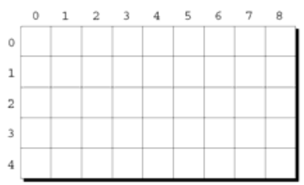
```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

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

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:

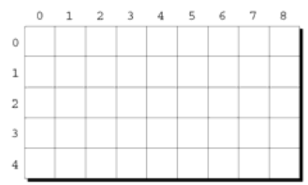
```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```



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}};
```



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.

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.

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.

Constant Arrays

- Advantages of declaring an array to be `const`:
  - Documents that the program won't change the array.
  - Helps the compiler catch errors.

- `const` isn't limited to arrays, but it's particularly useful in array declarations.

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).

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?

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).


```
bool in_hand[4][13] = {false};
```

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.

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`.

deal.c

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

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

#define NUM_SUITS 4
#define 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;
}
```


Variable-Length Arrays (C99)

- In C89, the length of an array variable
  - must be specified by a constant expression.

```
bool in_hand[4][13] = {false};
```

- In C99, however, it's sometimes possible to use an expression that's *not* constant.
- The `reverse2.c` program
  - —a modification of `reverse.c`—
  - illustrates this ability.

reverse2.c

```
/* Reverses a series of numbers using a variable-length
   array - C99 only */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("How many numbers do you want to reverse? ");
  scanf("%d", &n);

  int a[n];   /* C99 only - length of array depends on n */

  printf("Enter %d numbers: ", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
```



```
  printf("In reverse order:");
  for (i = n - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```


Variable-Length Arrays (C99)

- The array `a` in the `reverse2.c` program
  - is an example of a ***variable-length array*** (or ***VLA***).
- The length of a VLA
  - is computed when the program is executed.
- The chief advantage of a VLA is that a program
  - can calculate exactly how many elements are needed.
- If the programmer makes the choice,
  - it's likely that the array will be too long (wasting memory) or too short (causing the program to fail).

Variable-Length Arrays (C99)

- The length of a VLA doesn't have to be specified by a single variable. Arbitrary expressions are legal:

```
int a[3*i+5];
int b[j+k];
```

- Like other arrays, VLAs can be multidimensional:

```
int c[m][n];
```

- Restrictions on VLAs:
  - Can't have static storage duration (discussed in Chapter 18).
  - Can't have an initializer.