Concepts

7.1 Collections

7.2 Reading

7.3 Sorting

7.4 Functions

7.5 Two dimensions

7.6 Initializers

7.7 Pointers

7.8 Strings

7.9 Case study

7.10 Arrays of strings

7.11 Program arguments

Summary

Programming, Problem Solving, and Abstraction

Chapter Seven

Arrays

Lecture slides prepared by Alistair Moffat

Chapter 7

Concepts

7.1 Linear collections of like objects

7.2 Reading in to an array

7.3 Sorting an array

7.4 Arrays and functions

7.5 Two dimensional arrays

7.6 Array Initializers

7.7 Arrays and pointers

7.8 Strings

7.9 Case study

7.10 Arrays of strings

7.11 Program arguments

Summary

Chapter 7 – Concepts

- Large collections of items.
- Algorithms for data transformations.
- Sorting.
- Arrays, pointers, and functions.
- Strings and arrays of strings.

7.1 Linear collections of like objects

An array is a collection of same-type variables laid out in memory in a regular pattern, with the individual objects in the collection identified by their ordinal position.

Indexing starts at offset zero. When `N` is 5, the array elements are `A[0]`, `A[1]`, `A[2]`, `A[3]`, and `A[4]`.

- `array1.c`

7.1 Linear collections of like objects

If an array has `n` initialized elements then the recipe

```
    for(var=0;var<n;var++)
```

processes them one by one.

The only operation that can be applied to the array as a whole is the application of a subscript via the “`[]`” operator.

Multiple arrays can be declared in a program, of different sizes and different underlying types.

7.1 Linear collections of like objects

A `#define` should be used to establish the array size.

The same symbolic constant, or a variable known to be less than it, is used everywhere the array is manipulated.

Over-declaring is normally fine – allocating 10,000 entries when only 100 might sometimes be used is not an issue.

Stepping outside the declared bounds of an array is a common mistake and does not result in an immediate run-time error – beware.

7.1 Exercise 1

What does this loop do?

```
A[0] = 1;
for (i=1; i<MAX; i++) {
    A[i] = i*A[i-1];
}
```

7.1 Exercise 2

What about this one???

```
A[0] = 2;
for (p=1,n=3; p<MAX; n++) {
    prime = 1;
    for (i=0; i<p; i++) {
        if (n%A[i]==0) {
            prime = 0;
            break;
        }
    }
    if (prime) {
        A[p++] = n;
    }
}
```

7.2 Reading into an array

When reading in to an array, the bounds must be respected.

A “buddy variable” records how many values have been read, and stays with the array through subsequent processing.

Something has to be done with extra input; it shouldn’t be silently discarded.

PPSAA

Concepts

7.1 Collections

7.2 Reading

7.3 Sorting

7.4 Functions

7.5 Two dimensions

7.6 Initializers

7.7 Pointers

7.8 Strings

7.9 Case study

7.10 Arrays of strings

7.11 Program arguments

Summary

7.2 Reading into an array

```
printf("Enter as many as %d values, ^D to end\n",
        maxvals);
n = 0; excess = 0;
while (scanf("%d", &next)==1) {
    if (n==maxvals) {
        excess = excess+1;
    } else {
        A[n] = next;
        n += 1;
    }
}
printf("%d values read into array", n);
if (excess) {
    printf(", %d excess values discarded", excess);
}
printf("\n");
```

7.3 Sorting an array

One immediate task may be to sort an array.

Sort these numbers into order:

12 34 55 43 66 61 18 16 29 33 19

You will have followed a defined process, or algorithm.

There may be multiple algorithms for solving a given problem, each with different advantages and disadvantages.

7.3 Sorting an array

One simple algorithm is called insertion sort:

- invariant: from $A[0]$ to $A[i-1]$ is sorted;
- setting i to 1 initializes the invariant;
- the element $A[i]$ is considered, and by swapping elements one position to the right, as required, the correct spot to insert it is found;
- then i can be incremented, and the invariant still holds;
- when i reaches n, the n elements in A are sorted.

(*Note*: in the first edition the example sorting algorithm was bubble sort. You need to learn insertion sort.)

7.3 Sorting an array

When applied to the array {22, 14, 17, 42, 27, 28, 23}:

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Initially	22	14	17	42	27	28	23
After i=0	**22**	14	17	42	27	28	23
After i=1	**14**	22	17	42	27	28	23
After i=2	14	**17**	22	42	27	28	23
After i=3	14	17	22	**42**	27	28	23
After i=4	14	17	22	**27**	42	28	23
After i=5	14	17	22	27	**28**	42	23
After i=6	14	17	22	**23**	27	28	42

7.3 Sorting an array

Insertion sort is a relatively bad algorithm – inserting $A[i]$ might require i swaps (what input causes this?), making the total number of swaps as large as

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \approx n^2/2\,.$$

The worst case (and average case) behavior of insertion sort is quadratic in the number of items being sorted.

7.3 Sorting an array

If the number of array elements doubles, the running time is likely to increase by a factor of four.

At 100×10^6 operations per second, sorting 10,000 items takes around half a second.

But sorting 1,000,000 items will take 5,000 seconds.

And sorting 10,000,000 items will take 500,000 seconds, or six days!

Better sorting algorithms, efficient for billions of items, are introduced in Chapter 12.

7.4 Arrays and functions

If `A` is an array, then `A[0]` is its first element, and `&A[0]` the address of its first element.

The array name `A` is a pointer constant that has the same value as `&A[0]`, and can be used anywhere that a pointer would be.

So when an array is passed to a function, it receives a pointer to the first element of the array.

7.4 Arrays and functions

The function can declare the argument variable to be either a pointer, or an undimensioned array.

That pointer can then be used within the function to access the elements of the original array.

With array arguments side effects are always possible.

7.4 Arrays and functions

A whole program that brings all these ideas together:

- `insertionsort.c`

The buddy variable always accompanies the array.

7.4 Exercise 3

Write a function `is_sorted(int A[], int n)` that returns “true” if the array `A[0..n-1]` is in sorted order.

7.4 Arrays and functions

The typedef facility helps keep track of types, and makes programs easier to manage.

```
typedef double vector_t[SIZE];

void
vector_add(vector_t A, vector_t B, vector_t C,
           int n) {
    int i;
    for (i=0; i<n; i++) {
        C[i] = A[i] + B[i];
    }
}
```

7.4 Arrays and functions

All important recurring types (simple and compound) should be named using `typedef`. It is another way of enhancing modifiability.

The new types can be used for variable and argument declarations.

7.5 Two-dimensional arrays

Any type can be used as the base type of an array, including another array.

```
int X[10];
int (Y[5])[10];
```

Here `X` is an array of ten `int`, and `X[0]` is an `int`.

Similarly, `Y` is an array of five “array of ten `int`”, meaning that `Y[0]` must be an “array of ten `int`”.

7.5 Two-dimensional arrays

`(Y[0])[0]` is the first `int` in the first row, `(Y[0])[9]` the last.

`Y[4]` is the last row, and also has 10 `int` elements, named `(Y[4])[0]` to `(Y[4])[9]`.

The precedence rules allow the parentheses to be dropped: `(Y[4])[0]` can be written as `Y[4][0]`. (But not `Y[4,0]`.)

Nested `for` loops are the natural control structure for two-dimensional arrays.

- `twodarray.c`

7.5 Two-dimensional arrays

Only the dominant dimension can be omitted in the argument type declaration. Subsequent dimensions must be provided so that address calculations can be done correctly.

Use hierarchical typedefs and matching hierarchical functions.

- `matrixadd.c`

7.5 Exercise 4

Write a function `sqmatrix_mult(sqmatrix A, sqmatrix B, sqmatrix C, int n)` that calculates the matrix product `C` $=$ `A` $\times$ `B`, assuming that each matrix is `n` by `n`.

7.5 Two-dimensional arrays

When manipulating two-dimensional arrays:

- Both dimensions might be fixed, and no auxiliary size variables are needed, or
- Each row of the matrix might be full, but the number of rows might be variable, or
- The number of rows and columns in use is the same, and a single buddy variable is required, or
- Two buddy variables are in use, indicating the numbers of rows and columns that have been initialized.

7.5 Two-dimensional arrays

Higher dimensional arrays can be similarly declared.

Slices of reduced dimensionality are obtained as each subscript is supplied from the left.

The space required might grow quickly. The declaration `double Z[200][200][200][200]` involves 1.6 billion elements, and 12 GB of memory.

Even initializing that much memory is slow.

7.6 Array initializers

Arrays can be initialized on declaration:

```
#define MONTH_ARRAY 13
int month_days[MONTH_ARRAY] =
    {0,31,28,31,30,31,30,31,31,30,31,30,31};
```

Missing values at the end are assumed to be zero.

The compiler can be left to infer the array size if an initializer is given:

```
int month_days[] =
    {0,31,28,31,30,31,30,31,31,30,31,30,31};
```

7.6 Array initializers

A sentinel value can also be used:

```
int data_vals[] =
    {17,5,8,16,68,54,33,-1};

for (i=0; data_vals[i]!=-1; i++) {
    /* process data_vals[i] */
}
```

If new values are added prior to the sentinel, the loop automatically adjusts.

The sentinel may not be a valid data value. Ever!

PPSAA

Concepts

7.1 Collections

7.2 Reading

7.3 Sorting

7.4 Functions

7.5 Two dimensions

7.6 Initializers

7.7 Pointers

7.8 Strings

7.9 Case study

7.10 Arrays of strings

7.11 Program arguments

Summary

7.6 Array initializers

Multi-dimensional arrays are initialized by supplying a set of initializers, one per row:

```
int month_days[2][13]
    = {{0,31,28,31,30,31,30,31,31,30,31,30,31},
       {0,31,29,31,30,31,30,31,31,30,31,30,31}};

month_len = month_days[is_leap_year(yyyy)][mm];
```

7.7 Arrays and pointers

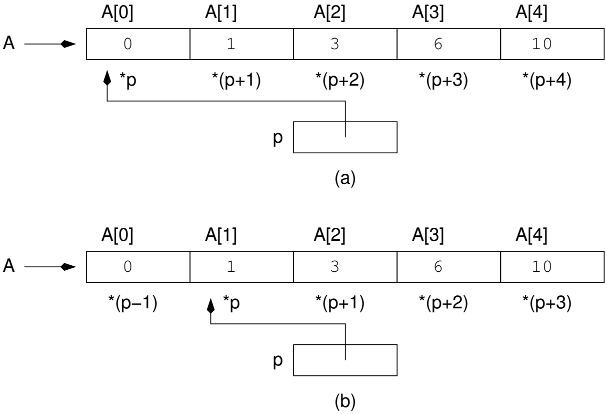
An array name is a pointer constant the value of which is the address of the first item in the array.

Pointer variables can step through an array, using pointer arithmetic.

If `p` is of type “pointer to *T*”, then `p+1` is also of type “pointer to *T*”, and points to the next unit of type *T* following the one pointed at by `p`.

- `pointer4.c`

7.7 Arrays and pointers



After `p=A`, and then after `p=p+1`.

7.7 Arrays and pointers

Pointers can be assigned and compared.

Be careful: there is a big difference between `p==A` and `*p==*A`; and between `p=A` and `*p=*A`

Pointers can also be subtracted to get an `int`.

7.7 Arrays and pointers

When the array is two-dimensional, things get complicated.

Adding one to the pointer shifts it by the number of bytes required to store an object of the corresponding base type.

The base type of a two-dimensional array is a one-dimensional array.

Easy to make mistakes! Best to just use double subscripts.

7.8 Strings

Arrays of `char` are used to store strings.

C requires that every string be terminated by a null byte character, written as “`\0`”. When interpreted as an `int` the null byte has the value 0.

The null byte acts as a sentinel, and is used by the string handling functions in the library described by `string.h`.

Space for it must be included in the array.

7.8 Strings

Consider these declarations:

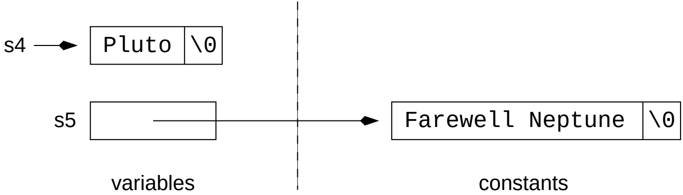
```
char s1[5] = {’H’, ’e’, ’l’, ’l’, ’o’};
char s2[6] = {’W’, ’o’, ’r’, ’l’, ’d’, ’\0’};
char s3[100] = "Goodbye";
char s4[] = "Pluto";
char *s5 = "Farewell Neptune";
```

The first four are arrays of `char`. But `s1` is not null-terminated, and is not a string.

Arrays `s2`, `s3`, and `s4` are strings. Arrays `s2` and `s4` are sized for the strings they contain; `s3` is over-declared.

7.8 Strings

Variable s5 is a pointer. The compiler allocates an initialized array elsewhere in memory, and initializes the pointer variable s5 to point at it.



7.8 Strings

The assignment `s5=s4` is legal, since `s5` is a variable. But changing `s5` makes the original string inaccessible.

The assignment `s5[0]=’a’` might result in a memory access error, since the underlying array is not your variable.

7.8 Strings

Pointers can be used to access strings:

- `string1.c`

7.8 Strings

String pointers indicate the first character of the string. The string continues through until the next null byte.

String operations never look to the left of the character pointer they are passed.

The library described by `string.h` includes functions for string copying, comparison, and concatenation.

- `strcpy.c`

7.8 Exercise 5

Write the function `strlen(char *s)` that returns the length (in characters) of the string indicated by `s`.

If you find it easier to think in terms of arrays, regard the argument as being of type `char s[]`.

PPSAA

Concepts

7.1 Collections

7.2 Reading

7.3 Sorting

7.4 Functions

7.5 Two dimensions

7.6 Initializers

7.7 Pointers

7.8 Strings

7.9 Case study

7.10 Arrays of strings

7.11 Program arguments

Summary

7.9 Case study

Design and implement a program that reads text from the standard input, and writes a list of the distinct words that appear. Words may be limited to a maximum of 10 alphabetic characters. Assume that as many as 1,000 distinct words might appear.

Each word should only be written once, when it first appears. Subsequent appearances must not be printed again. So need an array of character strings to store words.

7.9 Case study

What function(s) make sense? What tasks might be useful in other programs?

- `getword.c`
- `words.c`

7.9 Case study

There are many points to note:

- The use of `ctype.h` and function `isalpha`; and of `string.h` and the functions `strcpy` and `strcmp`.
- The use of `EOF` as a return value from the function.
- The use of conditional evaluation in the second loop in function `getword`, to ensure that operations only take place if they are “safe”.
- The insertion of the null byte at the end of the word.
- The use of `typedef`.
- The use of linear search, and of a flag variable.
- The bounds checking before the array assignment.

Concepts
7.1 Collections
7.2 Reading
7.3 Sorting
7.4 Functions
7.5 Two dimensions
7.6 Initializers
7.7 Pointers
7.8 Strings
7.9 Case study
7.10 Arrays of strings
7.11 Program arguments
Summary

7.9 Exercise 6

Modify function `getword` so that uppercase letters in words are mapped to the equivalent lower-case letters.

Modify function `getword` so that numeric strings are also permitted, but so that alphabetic words do not contain digits, and digit-words do not contain alphabetic characters.

7.9 Exercise 7.16

Modify the `main` so that the frequency of each word is printed as well.

Modify the `main` so that the ordering in the output is by decreasing word frequency.

7.10 Arrays of strings

The case study uses a “rectangular” character matrix to store the array of strings. If the dimensioning is conservative, a large amount of space might be wasted.

For static collections of strings, a ragged matrix can be declared using an array of initialized string pointers.

7.10 Arrays of strings

The pointer constant `NULL` has the integer value zero.

It is a valid pointer value that represents an invalid address. All unused pointers should be initialized to `NULL`.

`NULL` can also be used as a sentinel in an array of pointers, and as a guard in loops that iterate over such arrays.

7.11 Program arguments

The integer `argc` is a count of the number of strings that were on the command-line that executed this program.

The array `argv` contains pointers to strings, one per argument. Pointer `argv[0]` is thus always the name of the program currently executing.

- progargs.c

PPSAA

Concepts

7.1 Collections

7.2 Reading

7.3 Sorting

7.4 Functions

7.5 Two dimensions

7.6 Initializers

7.7 Pointers

7.8 Strings

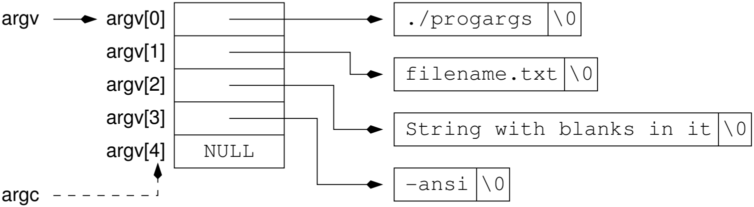
7.9 Case study

7.10 Arrays of strings

7.11 Program arguments

Summary

7.11 Program arguments



7.11 Program arguments

Another way that program arguments might be used:

```
int n=DEFAULT_N;
if (argc>1) {
    n = atoi(argv[1]);
}
printf("Using n=%d\n", n);
```

Function `atoi` converts a string to an integer, in the same way that `scanf("%d",...)` does.

Chapter 7 – Summary

- Arrays allow large volumes of data to be managed.
- Arrays can also be accessed using pointers.
- In a function, an array argument is always accessed via a passed pointer.
- Array types can be hierarchically composed.
- Strings are stored in arrays of `char`.
- Arrays of strings are a standard C data structure.