Notes 8.0: Arrays

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2014 session 1

Matrices

Here is an example of a finite one-dimensional matrix A:

$$\begin{pmatrix} 1 & 3 & 5 & 7 & 9 \end{pmatrix}$$

Indexing the elements of A starting from 0, $A(0)$ refers to 1, $A(1)$ to 3, etc.

Here is an example of a finite two-dimensional matrix B:

$$\begin{pmatrix} 1 & 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 5 & 7 & 9 & 11 \end{pmatrix}$$

B has three rows and five columns. Indexing the rows and columns of B starting from 0, $B(0,0)$ refers to 1, $B(0,3)$ to 7, $B(1,2)$ to 6, etc.

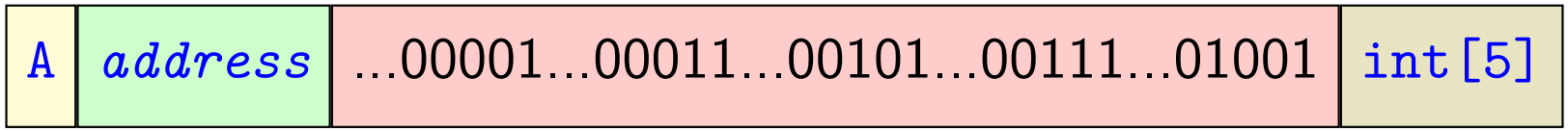
Parallelepipeds allow us to geometrically represent 3-dimensional finite matrices; there is often a need to consider N-dimensional finite matrices with $N \geq 4$.

One-dimensional arrays

To represent a one-dimensional array, C offers one-dimensional arrays; the matrix A would be represented by an array declared and initialised using a statement of the form

```
int A[5] = {1, 3, 5, 7, 9};
```

which corresponds to the following data item:



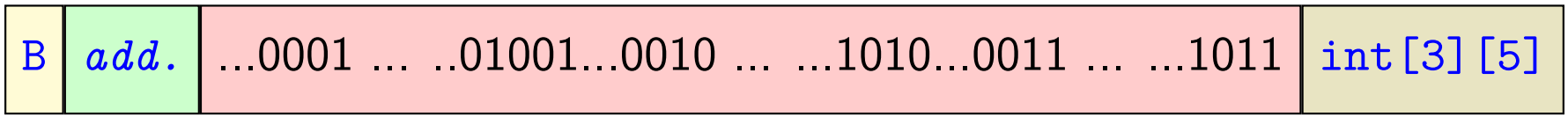
`A[0]` would evaluate to `1`, `A[1]` to `3`, etc.

Two-dimensional arrays (1)

To represent a two-dimensional array, C offers two-dimensional arrays; the matrix B would be represented by an array declared and initialised with a statement of the form

```
int B[3][5] = {{1, 3, 5, 7, 9},
               {2, 4, 6, 8, 10},
               {3, 5, 7, 9, 11}};
```

which corresponds to the following data item:



`B[0][0]` would evaluate to `1`, `B[0][3]` to `7`, `B[1][2]` to `6`, etc.

Two-dimensional arrays (2)

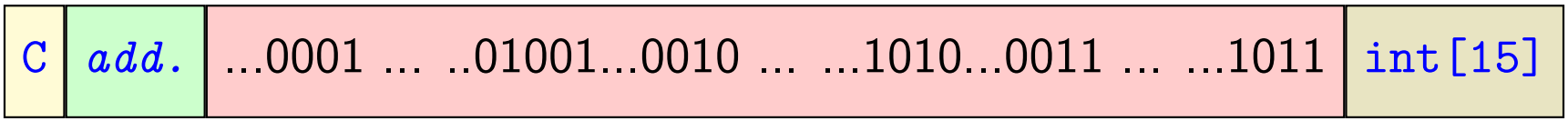
The one-dimensional matrix C defined as

$$\begin{pmatrix} 1 & 3 & 5 & 7 & 9 & 2 & 4 & 6 & 8 & 10 & 3 & 5 & 7 & 9 & 11 \end{pmatrix}$$

would be represented by an array declared and initialised with a statement of the form

```
int C[15] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10, 3, 5, 7, 9, 11};
```

which corresponds to the following data item:



The data items that represent B and C have the same internal representation of data.

N-dimensional arrays

Arrays of dimension higher than 2 are often useful. For instance,

```
int D[4][3][2] = {{{1, 1}, {3, 4}, {5, 6}},
                  {{7, 8}, {9, 10}, {11, 12}},
                  {{13, 14}, {15, 16}, {17, 18}},
                  {{19, 20}, {21, 22}, {23, 24}}};
```

would define and initialise a 3-dimensional array, and `D[0][2][1]` would evaluate to 6.

More on array initialisation (1)

Though not recommended, the previous declaration can be written as

```
int B[4][3][2] = {1, 1, 3, 4, 5, 6,
                  7, 8, 9, 10, 11, 12,
                  13, 14,15, 16, 17, 18,
                  19, 20, 21, 22, 23, 24};
```

and even as

```
int B[][3][2] = {1, 1, 3, 4, 5, 6,
                 7, 8, 9, 10, 11, 12,
                 13, 14,15, 16, 17, 18,
                 19, 20, 21, 22, 23, 24};
```

Indeed, the 3 and 2 give the compiler enough clues to correctly retrieve the curly braces and compute the missing dimension.

More on array initialisation (2)

But it won't accept a declaration that starts with either `int B[4][][2]` or `int B[4][3][]`.

As for any other variable, global arrays are automatically initialized but local arrays are not.

Even though the use of the constant `SIZE` minimises the risk of making an error on the array bounds, a better alternative is to write the for statements as follows (parentheses could surround the arguments of `sizeof`, but they are optional as those arguments are not types):

```
for (int i = 0; i < sizeof array1  / sizeof array1[0]; ++i)
    ....
for (int i = 0; i < sizeof array2 / sizeof array2[0]; ++i)
    ....
```

- `sizeof array1` is the size, in bytes, of the whole array `array1`;
- `sizeof array1[0]` is the size, in bytes, of one array element.

More on array initialisation (3)

If the number of initialisers is smaller than the number of array elements, then the default initialisation value is used for the remaining elements. For instance,

- `int a[4][3] = {{1, 2}, {3}, {4, 5}};`
  initialises `a` to

  `{{1, 2, 0}, {3, 0, 0}, {4, 5, 0}, {0, 0, 0}}`
- `int a[4][3] = {1, 2, 3, 4, 5};`
  initialises `a` to

  `{{1, 2, 3}, {4, 5, 0}, {0, 0, 0}, {0, 0, 0}}`

The last form is useful to initialise all elements of an array to the default value: with `0` as default value, an assignment of `{0}` does the job for both unidimensional and multidimensional arrays.

Variable length arrays

C99 has introduced variable length arrays, whose sizes do not need to be known at compile time but can be computed at run time. For instance,

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int size;
    printf("Enter size of array: ");
    scanf("%d", &size);
    if (size < 1)
        return EXIT_FAILURE;
    int a[size];
    return EXIT_SUCCESS;
}
```

would create an array of 12 `int`s if the user entered `12` when prompted to give a value to `size`. Variable size arrays cannot be initialised: hence the statement `int a[size];` cannot be replaced by `int a[size] = {0};`.

Assignment of array values

After an array has been declared, values can only be assigned to its elements by using indexes:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int i, arr1[5];
    for (i = 0; i < 5; ++i)
        arr1[i] = 2 * i;          // OK
    int arr2[] = {1, 2, 3, 4, 5};
    int arr3[5];
    arr3 = {3, 4, 5, 6, 7};       // NOT OK
    int arr4[5] = arr1;           // NOT OK
    return EXIT_SUCCESS;
}
```

An example: shuffling cards (1)

The program riffle_shuffle.c

- initialises a brand new deck of 52 cards, whose cards are ordered
  - starting with Ace of Hearts through King for Hearts,
  - followed by Ace of Clubs though King of Clubs,
  - followed by King of Diamonds through Ace of Diamonds,
  - followed by King of Spades through Ace of Spades;
- displays the deck from the first card (at the top) to the last one (at the bottom);
- riffle shuffles the deck and displays it, three times;
- riffle shuffles the deck five more times and displays it at the end.

An example: shuffling cards (2)

The program makes use of user-defined functions. It also makes use of three functions from the standard library.

- The function `rand()`, whose prototype is given in `<stdlib.h>`, returns a random integer number in the range 0 to `RAND_MAX`, defined in `<stdlib.h>`.
- The function `srand()`, whose prototype is also given in `<stdlib.h>`, takes an `unsigned int` as argument. It is called in order to provide a seed to `rand()`.
- The function `time()`, whose prototype is given in `<time.h>`, is used here to provide a new argument to `srand()` every time the program is run: when given `NULL` as an argument, it returns the number of seconds that have elapsed since the 1st of January 1970.

An example: shuffling cards (3)

It makes use of wide characters from the Unicode character set, of type `wchar_t`, together with one of the functions that can operate on them, namely, `putwchar()`, declared in the header file `wchar.h`.

The header file `locale.h` also needs to be included so that the statement

```
setlocale(LC_CTYPE, "");
```

can be executed, which is necessary to get the desired output.

It uses the `typedef` directive to give evocative names to suits, colours, and the type used to model a deck, namely, a 1-dimensional array of 52 `int`s.

The functions `/` and `%` used with a divisor of `13` allow one to easily obtain the colour and the rank of a card from its index.

An example: shuffling cards (4)

It is also our first example of a program split over many files, with

- one header file `deck.h` containing the `typdedef` directives and the prototypes of the functions that perform interesting operations on a deck of card, which could be reused;
- one file `deck.c` in which those functions are implemented;
- one file `riffle_shuffle.c` with the `main()` function that includes `deck.h` and that tests the functions declared in the latter through simulation.

An example: shuffling cards (5)

In the header of `riffle_shuffle.c`, the two lines

```
* Other source files, if any, one per line, starting on the next line:        *
*        deck.c                                                               *
```

allow one to compile the program within Emacs when issuing `\C-c p` from that buffer (`deck.c` could contain two similar lines, with `riffle_shuffle.c` replacing `deck.c`, to also be able to compile the program from the buffer that displays the contents of `deck.c`).

The first line is automatically included in the template created when the user tries and open a new file with `\C-c o` and answers `no` to the question

```
Do you want a template for a one file rather than a multifile program?
```

An example: the game of life

The game of life, implemented in life.c, (very crudely) models how cells come to life and die, as a function of surrounding cells.

The game is played on a grid, here with 8 rows and 8 columns.

The rules of the game are the following.

- A cell survives to the next generation if it has two or three neighbours.
- A cell dies from overcrowding if it has at least four neighbours.
- A cell dies from loneliness if it has less than two neighbours.
- A cell is born into an empty position if it has exactly three neighbours.

The program makes use of two global two-dimensional arrays.

Character arrays

Text can be stored in a character array.

To indicate the end of text, the null character, represented by `'\0'`, whose value is 0, is used.

String constants are character arrays; their contents is surrounded by a pair of double quotes.

For instance, the string constant `"hello\n"` is represented by a character array of length 7, namely:

h	e	l	l	o	\n	\0

When given a character array as argument, `printf()` uses the `%s` format specifier, expects to find a null character in the array, and prints out all characters up to the first `\0`.