Arrays

CSC 240 - Data structures with C++ - Lyon College, FA24

Marcus Birkenkrahe

August 5, 2024

Contents

1 Overview **2**

2 Problem Descriptions, Solution Process, and Coding Rules **3**

3 Arrays: Reversing a Series of Numbers **5**

4 Solution **5**

5 Notes **7**

6 Arrays: Checking a Number for Repeated Digits **9**

7 Solution **9**

8 Notes **11**

9 Debugging with `gdb` **12**

10 Review questions **13**

11 Practice exercises **15**

12 Programming assignments **15**

13 Glossary **16**

14 Summary **17**

15 Sources **18**

1 Overview

We review C by continuing with derived data types that you should already know: arrays, pointers, functions. We review by coding and analyzing code.

In this section, we focus on one-dimensional array data structures. Multi-dimensional arrays are less important in C because *arrays of pointers* are a more flexible way to store multi-dimensional data.

Objectives:

- Tackling problems, solutions, and coding rules
- Different ways of developing and running code
- One-dimensional arrays, `for` loops, `scanf` and `printf`
- Use of macros and `sizeof` to set/read array lengths
- Variable-sized arrays (new!) and new basic data types
- Application: Reversing series of numbers
- Application: Checking numbers for repeated digits
- Code along and solve different exercises
- Review questions, glossary, and summary

Code along:

1. Open Emacs from the command line with `emacs &`
2. Create an Org-mode file with `C-x C-f 2_arrays.org`
3. Create a C code chunk with `<s <TAB> C`
4. Add header arguments: `:main yes :includes <stdio.h> :results output`
5. Write and run a "hello world" test program: `puts("hello world");`
6. Keep running chunks with `C-c C=c` to check for syntax errors.
7. If any of these steps don't work for you, let me know!

See my YouTube channel (`@Literate Programming with Emacs and Org-mode`) for a short literate programming tutorial: tinyurl.com/first-lit-prog.

2 Problem Descriptions, Solution Process, and Coding Rules

Problem Components

Each problem begins with three components:

1. The problem context as a short description including potential constraints.
2. The problem input in the form of test cases which must be passed.
3. The problem output in the exact form required (optionally with time limit[1]).

Solution Components

Each solution includes:

1. A short description how we tackle the problem.
2. Minimal comments in the source code.
3. Functions should be accompanied by a short summary at the top:
   - Name and purpose of the function
   - Return type and parameters
   - Anything noteworthy (for example, edge cases, or ways to break the code, or version dependencies)
4. Program organization:
   (a) Preprocessor directives
   (b) Type definitions
   (c) Declarations of external variables
   (d) Prototypes for functions other than `main`
   (e) Definition of `main`
   (f) Definitions of other functions

[1] "Time limit" is important if we're looking for the most efficient code, which depends on the size of the data and the exact use case, and is highly influenced by the chosen data structures.

5. Review (not in the code blocks):
    - Anything you learnt that you may have forgotten
    - Additional code snippets for illustration
    - Open questions
    - Short summary and/or glossary of terms/commands
    - Edge cases (special situations or values)

General Problem-Solving Techniques

These should always be part of your problem-solving approach:

1. Always have a plan
2. Restate the problem
3. Divide the problem
4. Start with what you know
5. Reduce the problem
6. Look for analogies
7. Experiment
8. Don't get frustrated
9. Review your solution
10. Develop an alternate solution

Coding Rules

There are a few Coding rules:

1. You should always write your own code (not just copy and paste) though you may use secondary sources to help you.
2. You should stay away from using AI for code generation, debugging, etc. until you've spent an appropriate length of time trying to crack a problem yourself[2].

[2] An "appropriate amount of time trying to solve a problem" is very personal - if you really want to learn this stuff, you need to spend time mulling things over, perhaps for hours or even days. This usually implies developing strategies for finding and absorbing technical information - e.g. function or compiler documentation.

3. You should be able to explain every part of your code.

3 Arrays: Reversing a Series of Numbers

- **Problem:** Prompt the user to enter a series of numbers, then write the numbers in reverse order. Store the numbers in an array and use a preprocessing directive to set the size of the array. In the `main` function, initialize the array's elements to zero.

- **Input** and **Output**:

```
Enter 10 numbers: 34 82 49 102 2 94 23 11 50 31
In reverse order: 31 50 11 23 94 2 102 49 82 34
```

- Ask yourself if you could write this program right away or not! Whatever the answer, let's do it together from scratch.

4 Solution

- **Strategy:** The numbers are stored in an array of N = 10 elements. N is defined at the start with `#define`. The array is initialized with zeros. The numbers are retrieved from standard input with `scanf` counting up with a `for` loop, and printed in reverse order with `printf` counting down with a `for` loop.

- **Program** `reverse.c`:

```
/*************************************************************/
// reverse.c: reverse a series of numbers entered by the user
// Store numbers as int array with scanf in a for loop
// Print numbers in reverse order with printf in a for loop
/*************************************************************/
#include <stdio.h> // include input/output header file
#define N 10  // define constant array length with directive

int main() // main program
{
  int i;           // declare loop variable
  int a[N] = {0};  // macro definition (uses constant N)
  printf("Enter 10 numbers: "); // ask for input
```

```
  for ( i = 0; i < N; i++) { // loop over array (counting up)
    scanf("%d", &a[i]); // get the i-th array element from stdin
    printf("%d ", a[i]); // print the i-th array element to stdout
  }
  puts("");  // add new line
  printf("In reverse order: ");
  for ( i = N-1; i >= 0; i--) { // loop over array (counting down)
    printf("%d ", a[i]);
  }
  puts("");
  return 0; // confirm program ran successfully
} // end of main program


Enter 10 numbers: 9 16 47 82 4 66 12 3 25 51
In reverse order: 51 25 3 12 66 4 82 47 16 9
```

- To compile and run this program, you have three options[3]:

  1. Add the header argument `:cmdline < input` where `input` is a text file containing the test input data. Then run the code block with `C-c C-c`. You can create an input file in a shell code block.
  2. Tangle the source code for compilation, debugging and execution on the command-line, add the header argument `:tangle reverse.c`, tangle with `C-u C-c C-v t` and run it on the command-line shell with the chain command: `gcc reverse.c -o rev && ./rev`.
  3. You can also tangle the file as `reverse.c` (`C-u C-c C-v t`) and run it in a shell code block with the test input:

     ```
     gcc reverse.c -o rev
     echo 34 82 49 102 2 94 23 11 50 31 | ./rev

     Enter 10 numbers: 34 82 49 102 2 94 23 11 50 31
     In reverse order: 31 50 11 23 94 2 102 49 82 34
     ```

     Let's see how this works:

[3] If this was R, Julia or Python (interpreted rather than compiled languages), you'd have another open, namely opening the source file in a dedicated buffer with `C-c '` and then running all or part of it in the console (the R, Julia or Python shell), and returning to Org-mode with `C-c C-k`.

(a) `gcc reverse.c -o rev` compiles the file and creates an output file called `rev`

(b) `echo` prints its arguments (the test series) to stdout

(c) The pipe symbol `|` takes the output on its left and serves it as stdin on the right.

(d) `./rev` receives the input from the left and runs with it.

(e) The `./` is necessary for the shell to find the executable file `rev` in the current directory.

5 Notes

1. Is the program proofed against wrong input? Try to break it using the command-line executable entering characters or words instead, or leave out numbers.

2. You can use *variable-length arrays* if you don't want to fix the length of the array - but you cannot initialize it (since the length of the array is not known at compile-time):

```
int i, n;
printf("How many numbers do you want to reverse: ");
scanf("%d", &n);
int a[n];
printf("Enter %d numbers: ", n);
for ( i = 0; i < n; i++) {
  scanf("%d", &a[i]);
 }; puts("");
printf("In reverse order: ");
for ( i = n-1; i >= 0; i--) {
  printf("%d ", a[i]);
 }; puts("");
```

Tangle the source file `reverse2.c`, and on the command-line, run:

```
gcc reverse2.c -o rev2
echo 4 5 4 3 2 | ./rev2  # output: 2 3 4 5


How many numbers do you want to reverse: Enter 4 numbers:
In reverse order: 2 3 4 5
```

3. We can compute the length of an array `a` using the `sizeof` operator, which is useful if we don't know the length[4].

```
#define N 10
#define SIZE (int)(sizeof(a)/sizeof(a[0]))

int main (void)
{
  int i;
  int a[N];
  printf("SIZE = %d\n", SIZE);
  for (i = 0; i < SIZE; i++) {
    a[i] = i+1;
    printf("%d ", a[i]);
  }
  return 0;
}
```

```
SIZE = 10
1 2 3 4 5 6 7 8 9 10
```

4. Macros can also have parameters. Here are two examples:

```
#define MAX(x,y) ((x)>(y) ? (x):(y))
#define IS_EVEN(n) ((n)%2==0)

printf("Max value: %d\n", MAX(100,200)); // use parametrized macro
printf("Is 100 even? %d\n", IS_EVEN(100)); // use parametrized macro
```

```
Max value: 200
Is 100 even? 1
```

In the definition of `MAX`, `x > y ? x : y` is a short version of `if...else`:

[4] `sizeof(a)` returns the size of `a` in bytes as an unsigned `int`. If you divide by the byte-size of a single element, you get the number of elements. We use `(int)` to cast the unsigned `int` of the `sizeof` result, to avoid compiler warnings.

```
if (x > y)
   x
else
   y
```

You can also use macros to create aliases for commands you're tired of typing, like `printf("%d\n",i);`

```
#define PRINT_INT(n) printf("%d\n",n);

int i = 200, j = 100;
PRINT_INT(i/j);


2
```

6 Arrays: Checking a Number for Repeated Digits

- **Problem:** 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`:
- **Input** and **Output**:

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

- Ask yourself if you could write this program right away or not! Whatever the answer, let's do it together from scratch.

7 Solution

- **Strategy**: The program uses an array of Boolean values `digits_seen` to keep track of which digits 0-9 xappear in a number. Initially, every element of the array is `false`. When given a number `n`, the program examines its digits one at a time, storing each into the `digit` variable, and then using it as an index into `digit_seen`. if `digit_seen[digit]` is `true`, then `digit` appears at least twice in `n`. 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.

- **Program** `repdigit.c`:

```
/*********************************************************/
// repdigit.c: checks numbers for repeated digits.
// Input: number with (without) repeated digits.
// Output: Print "Repeated digit" or "No repeated digit."
/*********************************************************/
#include <stdbool.h> // defines ‘bool‘ type
#include <stdio.h>

int main(void)
{
  // variable declarations and initialization
  bool digit_seen[10] = {false}; // initialized to zeros = false
  int digit;
  unsigned long int n; // an unsigned long integer type

  // getting user input
  printf("Enter a number: ");
  scanf("%lu", &n);
  printf("%ld\n", n);

  // scan input number digit by digit
  while (n > 0) { // loop while n positive
    digit = n % 10; // example output: 28212 % 10 = 2
    if (digit_seen[digit]) // if true then digit repeats
break; // leave loop
    digit_seen[digit] = true;
    n /= 10; // example output: (int) (28212/10) = (int) 2821.2 = 2821
  } // finishes when (int) single digit / 10 = 0

  // print result
  if (n > 0) // found repeat digit before scanning whole number
    printf("Repeated digit\n");
  else  // n = 0 means scanning finished = all digits seen
    printf("No repeated digit\n");

  return 0;
}
```

- Testing:

```
gcc repdigit.c -o rep
echo 1987654321 | ./rep
```

```
Enter a number: 1987654321
Repeated digit
```

8 Notes

1. Make sure you understand the algorithm:
   (a) `digit_seen` is a Boolean vector of 10 values initialized to be `false` when none of the digits of `n` have been visited yet.
   (b) The first `digit` is `n % 10`, that is the remainder of dividing `n` by 10. For example for 28212 = 2821 * 10 + 2, 28212 % 10 = 2.
   (c) The array element corresponding to `digit = 2` is `digit_seen[digit] = digit_seen[2]`. It is 0 (`false`) if the digit has not been seen yet, and it is 1 (`true`) if it has been seen.
   (d) In the latter case (1), the `break` command leads out of the `while` loop, because the answer "Are there any repeated digits" has been answered.
   (e) In the former case (0), `digit_seen[true]` is now set to 1 (because the digit has been seen), and we move on to the next digit: `n / 10` removes the last digit, e.g. `28212 / 10 = 2821.2` but `int(2821.2) = 2821`, and the loop starts over for the next digit.
   (f) If the loop was left early because a repeated digit was found, `n > 0`, and `"Repeated digit"` is printed.
   (g) If the loop ran through all digits, no repeated digits were found, and `n=0` because the last digit divided by 10 is smaller than 10, hence its integer part is 0, and `No repeated digit` is printed.
2. You don't need to load `<stdbool.h>`, you can also `#define` Boolean values and use C's `typedef` keyword to create a synonym for previously defined types:

```
#define true 1
#define false 0
typedef int bool;
```

3. C has a number of different `int` types. On a 64-bit machine, `unsigned long int` can hold positive whole numbers up to 18,446,744,073,709,551,615. Do you know why that is?

   > The largest value is 2^64-1: 64 bits (or 8 words/bytes of 8 bit length) can be used to represent a value in binary number 0,1.
   >
   > For example, with 3 bits you can represent 2^3=8 values ranging from (000) to (111). Likewise, for an n-bit unsigned integer, the values range from 0 to 2^n-1. For n=64 that's the number given.
   >
   > Can you guess what the range of values will be for `long int` (which allows positive and negative integers)?[5]

```
#include <stdio.h>
#include <limits.h> // contains definition of ULONG_MAX

int main() {
  unsigned long int max_value = ULONG_MAX; // max value for unsigned
  // long int
  printf("The largest value for unsigned long int is: %lu\n", max_value);
  return 0;
}
```

```
The largest value for unsigned long int is: 18446744073709551615
```

4. To capture `unsigned long int` numbers, `scanf` requires the `%lu` format specification.

9 Debugging with `gdb`

- Check that you have the GNU debugger with `gdb --version`. If this does not work, you have to install `gdb` with `sudo apt install gdb`.

```
gdb --version
```

[5] The range of values for signed long integers is (-2^63-1,2^63-1) because one bit is lost for the sign, and there are now twice as many numbers, so the maximum value on a 64-bit machine is (2^63-1)/2 or 4,611,686,018,427,387,904.

```
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

- To debug files, you must compile the source code file with the additional flag `-g`:

  ```
  gcc repdigit.c -o rep -g
  ```

- For the debugging session, you need to open a shell in Emacs with `M-x shell` and run `gdb ./rep`.
- Once you're in the debugger, you can set break points with `break` (for example, `break main` will break at the start of `main`, or `break 19` will break at line 19 of the source code file), execute the file with `run`, go from statement to statement with `next`, get variables with `print` (e.g. `print n`), or all local variables with `info locals`.
- Let's do it together to understand the `repdigit.c` program better!
- The debugger is the more useful the more moving parts your program has, and we'll use it often when we start building larger programs with functions and data structures.
- For more information, see the `gdb(1)` man page (`M-x man gdb RET` in Emacs, or online).

10 Review questions

1. What constitutes a problem?
   (a) Problem context description with constraints.
   (b) Input in the form of test cases that must be passed.
   (c) Output in exact format with time limit (optional)
2. What does a solution include:
   (a) A solution strategy describing how you tackled the problem.

(b) Code comments, program and function headers.
(c) Standard program organization.
(d) Solution review with a discussion of edge cases.

3. What are the coding rules?

   (a) Write code yourself (minimize AI assistance)
   (b) Give yourself time to understand and solve a problem.
   (c) Be able to explain every part of your code.

4. What's a "macro" definition?

   A macro is a directive for the preprocessor to define a constant or a name, e.g. `#define PI 3.14`, which is replaced everywhere in the code. There are also *parametrized* or *function-like* macros like `#define PRINT_INT(n) print("%d\n",n)`.

5. How can you run a C program in an Org-mode code block (provided the program compiles) if the program requires you to give one character as input?

   Tangle the file with a header argument `:tangle main.c` and the keyboard commands `C-u C-c C-v t`; then create a shell (`bash`) code block. In this code block, compile the file and run it by piping the input to the executable, which must be called with the relative path:

```
gcc main.c -o main
echo 'a' | ./main
```

6. What is a variable-length array? What are the constraints on a VLA?

   A VLA is an array whose memory (aka length) is computed and allocated at runtime - for example, you can prompt the user for it. The primary restriction is that they cannot be initialized. Also, they can't have `static` storage duration.

7. What does the `(int)` do in the following preprocessor directive for an array `a`, and what is its output?

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

The directive defines `SIZE` so that wherever the compiler finds `SIZE`, it is replaced by `(int)(sizeof(a)/sizeof(a[0]))`. In this expression, the length of an array `a` is computed, and the result is converted into a (signed) integer because `sizeof` returns an `unsigned int`, to avoid compiler warnings.

Another question might be: what if I change the name of the array from `b` to `a`? Then the macro does no longer apply and must be changed, or a parametrized macro has to be used: `#define SIZE(array) (int)(sizeof(a)/sizeof(a[0]))` which works with any array name.

11 Practice exercises

1. Write a declaration of an array named `weekend` containing seven `bool` values. Include an initializer that makes the first and last values `true`; all other values should be `false`.

2. The Fibonacci numbers are 0,1,1,2,3,4,5,13,... where each number is the sum of the two preceding numbers. Write a program that declares an array named `fib` of length `20` and fills the array with the first 20 Fibonacci numbers.

3. Initialize a 2 x 2 identity matrix and print it:

   ```
   1 0
   0 1
   ```

   Hint: A two-dimensional array `a` is defined as `a[M][N]`.

12 Programming assignments

Submit your solution as an Org-mode file. The code should pass the test case, and the required output should be part of the Org-mode file. Since user input is required, compile and run the tangled file in a `bash` code block for problems 1-2, and for problem 3, tangle the file, and open a shell in Emacs (`M-x shell`) to compile and run it.

If you run into trouble, remember that you can debug your code with `gdb` if you compile the file with `-g`.

1. Modify the `repdigit.c` program, which checked a number for repeated digits so that it shows which digits (if any) were repeated.

   Sample input and output:

   ```
   Enter a number: 939577
   Repeated digit(s): 7 9
   ```

2. Modify the `repdigit.c` program, which checked a number for repeated digits so that it prints a table showing how many times each digit appears in the number:

   Sample input and output:

   ```
   Enter a number: 41271092
   Digit:        0  1  2  3  4  5  6  7  8  9
   Occurrences:  1  2  2  0  1  0  0  1  0  1
   ```

3. Modify the `repdigit.c` program, which checked a number for repeated digits so that the user can enter more than one number to be tested for repeated digits. The program should terminate when the user enters a number that's less than or equal to 0.

13 Glossary

	Term	Explanation
1	Array	A collection of contiguosly stored elements of the same type
2	Preprocessor	Directives that provide instructions to the compiler.
3	Macro	A fragment of code which is given a name.
4	`#define`	Used to define macros or constants.
5	`sizeof`	Operator that returns the size of a variable or datatype.
6	`scanf`	Function to read formatted input from stdin.
7	`printf`	Function to print formatted output to stdout.
8	Loop	A programming construct that repeats a block of code.
9	`for` loop	A control flow statement for specifying iteration.
10	Array length	The number of elements in an array.
11	Variable-length array	An array where the length is determined at runtime.
12	Compile-time	The period when source code is being compiled.
13	Runtime	The period when a program is running.
14	Tangle	Exporting source code from an Org-mode file.
15	`main` function	The entry point of a C program.
16	`puts`	Function to print a string followed by a newline.
17	`gcc`	GNU Compiler Collection, used to compile C programs.
18	Command-line	Interface for typing commands directly to the OS.
19	Shell (`bash`)	A program that interprets command-line input.
20	Input	Data provided to a program for processing.
21	Output	Data produced by a program.
22	External variable	Variable declared outside of any function.
23	Function	A block of code that performs a specific task.
24	Prototype	Declaration of a function's interface.
25	Edge case	A problem that occurs only in an extreme case
26	Debugging	The process of finding and resolving defects in software.
27	Compilation	The process of converting source code into executable code.
28	`unsigned long int`	Integer type that can hold a max value of 2^64-1
29	`long int`	Integer type that can hold a max value of (2^63-1)/2
30	`(%lu) =%ld`	Format specifier for (un)signed long integer values
31	`gdb`	GNU debugger

14 Summary

- Coding rules focus on understanding code, function summaries, and robustness.
- Proper program organization includes clear structure and minimal comments.

- The use of macros and `sizeof` ensures flexible and maintainable code.
- Using `sizeof` dynamically determines array size, enhancing robustness.
- Variable-sized arrays allow more flexible dynamic memory allocation.
- Besides `int` there are other data types like `unsigned long int`.
- Run Emacs code blocks in `bash` code blocks in the same Org-mode file.
- Debug your files step-by-step with `gdb` after compiling with `-g`.

15 Sources

- C Programming by K N King (W W Norton, 2008), chapter 8
- Think Like a Programmer by V Anton Spraul (NoStarch, chapter 1)