Problem Solving with C

Compiled by

M S Anand (anandms@pes.edu)

Text Book(s):

1. “How To Solve It By Computer”, R G Dromey, Pearson, 2011.
2. “The C Programming Language”, Brian Kernighan, Dennis Ritchie, 2nd Edition, Prentice Hall PTR, 1988.

Reference Book(s):

1. “Expert C Programming; Deep C secrets”, Peter van der Linden
2. “ The C puzzle Book”, Alan R Feuer

Designated initializers (C99)

It is often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values:
int a[15] = {0,0,29,0,0,0,0,0,0,7,0,0,0,0,48};

So, we want element 2 to be 29, 9 to be 7 and 14 to be 48 and the other values to be zeros. For large arrays, writing an initializer in this fashion is tedious.

C99's **designated initializers**

int a[15] = {[2] = 29, [9] = 7, [14] = 48};
Each number in the brackets is said to be a designator.

Besides being shorter and easier to read, designated initializers have another advantage: the order in which the elements are listed no longer matters. The previous expression can be written as:
int a[15] = {[14] = 48, [9] = 7, [2] = 29};

Designators must be constant expressions. If the array being initialized has length n, each designator must be between 0 and n-1. However, if the length of the array is omitted, a designator can be any non-negative integer. In that case, the compiler will deduce the length of the array by looking at the largest designator.

```
int b[] = {[2] = 6, [23]=87};
```

Because 23 has appeared as a designator, the compiler will decide the length of this array to be 24.

An initializer can use both the older technique and the later technique.

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

The initializer specifies that the array's first three elements will be 5,1 and 9, Element 4 will have value 3. The two elements after element 4 will be 7 and 2. Finally, element 8 will have the value 6. All elements for which no value is specified will default is zero.

Problem

Using arrays, write a program to check whether a given number has repeated digits.

Ex: 456754 (has repeated digits)
3456 (Does not have)

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

int main (void)
{
        long num;
        int digit;
        bool digit_seen [10] = {false};
        printf (“Enter a number:”);
        scanf (“%d”, &num);

        while (num > 0)
        {
                digit = num%10;
                if (digit_seen [digit])
                        break;
                digit_seen [digit] = true;
                num /= 10;
        }
        if (num > 0)
                printf (“Digits are repeated\n”);
        else
                printf (“No repitition\m”);
        return 0;
}
```

Using the sizeof operator with arrays

The sizeof operator executed on an array gives the size of the array in bytes:

```
char arr [20];
sizeof (arr) – results in 20 bytes
```

What about the following:

```
int iarr [20];
sizeof (iarr)   -  ??

long larr [40];
sizeof (larr)   -   ??
```

We can use sizeof to measure the size of an array element:

```
char carr [20];
sizeof (carr) – gives 20
sizeof (carr[0]) --  ??
```

What is the result of the following:

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

char m_array [20][30];
sizeof (m_array)  ---   ??
```

Constant arrays

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

The compiler will detect any attempt to modify the values and inform the user.

The “make” utility

Need

It is a good programming practice to keep the length of your functions to a reasonably small size (number of lines of source code). This will help in better understanding, control and maintainability.

Any well designed application will have a number of functions. It is a good practice to have these functions in multiple source files (.c files). This helps the programmer as well as anyone who might end up using these well defined functions in their programs.

How do we use gcc to compile multiple source files and generate an executable.

We can use gcc in the following way:

gcc –o(output file name) srce1.c srce2.c …..srcen.c

Imagine typing in the name of all the source file names correctly and making sure that only those source files which got modified since the last round of compilation only to be recompiled. This would be a tedious task and may become a real challenge as the number of source files increases. Here is where the “make” utility comes into picture.

What “make” does

“make” goes through a descriptor file starting with the **target** it is going to create. “make” looks at each of the target's **dependencies** to see if they are also listed as targets. It follows the chain of dependencies until it reaches the end of the chain and then begins backing out executing the commands found in each target's rule. Actually every file in the chain may not need to be compiled. Make looks at the time stamp for each file in the chain and compiles from the point that is required to bring every file in the chain up to date. If any file is missing it is updated if possible.

Make builds object files from the source files and then links the object files to create the executable. If a source file is changed only its object file needs to be compiled and then linked into the executable instead of recompiling all the source files.

You need to create a file named “makefile” which contains a few commands and then run the make command.

A makefile consists of rules with the following format:

```
target: dependencies ...
        commands
    ...
```

A Simple Example of a Makefile

```
=====   makefile starts here ===========================
myappl: srcfile1.o srcfile2.o srcfile3.o
        gcc -o myapp1 srcfile1.o srcfile2.o srcfile3.o -lm
srcfile1.o: scrfile1.c myinclude.h
        gcc -c srcfile1.c
srcfile2.o: scrfile2.c myinclude.h
        gcc -c srcfile2.c
srcfile3.o: scrfile3.c myinclude.h
        gcc -c srcfile3.c
clean:
        rm -f myapp1 srcfile1.o srcfile2.o srcfile3.o
==================  End of makefile ====================
```

As you can see, this program has three source files srcfile1.c, srcfile2.c and srcfile3.c. There is also a user defined header file “myinclude.h”

Note: You need to put a tab character at the beginning of each command line.

It is usually easiest to start with an existing makefile and edit it for a new application

Let's go through the example to see what make does by executing with the command “make myappl” and assuming the program has never been compiled.
make finds the target myappl and sees that it depends on the object srcfile1.o, srcfile2.o and srcfile3.o.
make next looks to see if any of the three object files are listed as targets. They are; so make looks at each target to see what it depends on. make sees that srcfile1.o depends on the files srcfile1.c and myinclude.h.
Now make looks to see if either of these files are listed as targets and since they aren't it executes the commands given in srcfile1.o's rule and compiles srcfile1.c to get the object file.
make looks at the targets srcfile2.o and srcfile3.o and compiles these object files in a similar fashion.
make now has all the object files required to make myappl and does so by executing the commands in its rule.
You probably noticed we did not use the target, clean, it is called a **phony target**

Arrays and Pointers – Differences

1. the sizeof operator
   a. sizeof(array) returns the amount of memory used by all elements in array
   b. sizeof(pointer) only returns the amount of memory used by the pointer variable itself
2. the & operator
   a. &array is an alias for &array[0] and returns the address of the first element in array
   b. &pointer returns the address of pointer
3. a string literal initialization of a character array
   a. char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
   b. char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)
4. Pointer variable can be assigned a value whereas array variable cannot be.
5. Arithmetic on pointer variable is allowed.

Pointers
Note

char* is a **mutable** pointer to a **mutable** character/string.

const char* is a **mutable** pointer to an **immutable** character/string. You cannot change the contents of the location(s) this pointer points to. Also, compilers are required to give error messages when you try to do so. For the same reason, conversion from const char * to char* is deprecated.

char* const is an **immutable** pointer (it cannot point to any other location) **but** the contents of location at which it points are **mutable**.

const char* const is an **immutable** pointer to an **immutable** character/string.

Debugging your programs

You would have probably used the “printf” function all over your code to try and debug your programs till now. There is a better tool available for this – the gdb (GNU debugger)

The next few slides present a quick introduction to gdb.

What is gdb?

1. “GNU Debugger”
2. A debugger for several languages, including C and C++
3. It allows you to inspect what the program is doing at a certain point during execution.
4. Errors like segmentation faults may be easier to find with the help of gdb.

Additional steps required during compilation to help you use gdb

Normally, you would compile a program like:

gcc [flags] <source files> -o <output file>

For example:

gcc – o outfile src1.c src2.c

Now you add a -g option to enable built-in debugging support (which gdb needs):

gcc [other flags] -g <source files> -o <output file>

Starting up "gdb"

Just try "gdb" or "gdb **outfile**" You'll get a prompt that looks like this:
(gdb)
If you didn't specify a program to debug, you'll have to load it in now:
(gdb) file **outfile**
Here, **outfile** is the program you want to load, and "file" is the command to load it.

gdb has an interactive shell, much like the one you use as soon as you log into the linux systems. It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.

Tip

If you're ever confused about a command or just want more information, use the "help" command, with or without an argument:
(gdb) help [command]

Running the program

To run the program, just use:
(gdb) run

This runs the program. If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.

If the program did have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum array region (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at sum-array-region2.c:12

What if bugs are present in the program?

Okay, so you've run it successfully. But you don't need gdb for that. What if the program isn't working?

Basic idea

Chances are if this is the case, you don't want to run the program without any stopping, breaking, etc. Otherwise, you'll just rush past the error and never find the root of the issue. So, you'll want to step through your code a bit at a time, until you arrive upon the error.

This brings us to the next set of commands. . .

Setting breakpoints

Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command “break.”
This sets a breakpoint at a specified file-line pair:

(gdb) break file1.c:6
This sets a breakpoint at line 6, of file1.c. Now, if the program ever reaches that location when running, the program will pause and prompt you for another command.

Tip
You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

You can also tell gdb to break at a particular function. Suppose you have a function my func:
int my func(int a, char *b);
You can break anytime this function is called:
(gdb) break my func

What next?

Once you've set a breakpoint, you can try using the run command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).

You can proceed onto the next breakpoint by typing "continue" (Typing run again would restart the program from the beginning, which isn't very useful.)
(gdb) continue

You can single-step (execute just the next line of code) by typing "step." This gives you really fine-grained control over how the program proceeds. You can do this a lot...
(gdb) step

Similar to “step,” the “next” command single-steps as well, except this one doesn’t execute each line of a sub-routine, it just treats it as one instruction.
(gdb) next

Tip
Typing “step” or “next” a lot of times can be tedious. If you just press ENTER, gdb will repeat the same command you just gave it. You can do this a bunch of times.

So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like the values of variables, etc. This might be useful in debugging. :)

The print command prints the value of the variable specified, and print/x prints the value in hexadecimal:
(gdb) print my var
(gdb) print/x my var

Setting watchpoints

Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a watched variable's value is modified. For example, the following watch command:

(gdb) watch my var

Now, whenever my var's value is modified, the program will interrupt and print out the old and new values.

Tip

You may wonder how gdb determines which variable named my var to watch if there is more than one declared in your program. The answer (perhaps unfortunately) is that it relies upon the variable's scope, relative to where you are in the program at the time of the watch. This just means that you have to remember the tricky nuances of scope and extent

Other useful commands

backtrace - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions)

where - same as backtrace; you can think of this version as working even when you're still in the middle of the program

finish - runs until the current function is finished

delete - deletes a specified breakpoint

info breakpoints - shows information about all declared breakpoints

More about breakpoints

Breakpoints by themselves may seem too tedious. You have to keep stepping, and stepping, and stepping. . .

Basic idea

Once we develop an idea for what the error could be (like dereferencing a NULL pointer, or going past the bounds of an array), we probably only care if such an event happens; we don't want to break at each iteration regardless.

So ideally, we'd like to condition on a particular requirement (or set of requirements). Using conditional breakpoints allow us to accomplish this goal. . .

Conditional breakpoints

Just like regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger. We use the same break command as before:

(gdb) break file1.c:6 if i >= ARRAYSIZE

This command sets a breakpoint at line 6 of file file1.c, which triggers only if the variable i is greater than or equal to the size of the array (which probably is bad if line 6 does something like arr[i]). Conditional breakpoints can most likely avoid all the unnecessary stepping, etc.