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

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.

Program (use of pointers)

Write a program that will read a string from the keyboard and display it after removing all spaces and punctuation characters. All operations should use pointers.

A few programs using two-dimensional arrays

Matrix multiplication

To multiply two matrices, the number of columns of first matrix should be equal to the number of rows to second matrix. This program displays the error until the number of columns of first matrix is equal to the number of rows of second matrix.

Transpose of a matrix

Transpose of a matrix is obtained by changing rows to columns and columns to rows. In other words, transpose of A[][] is obtained by changing A[i][j] to A[j][i].

An interesting question

How is a[i] == i[a]?

Compilers use pointer arithmetic internally to access array elements. And because of the conversion rules that apply to the binary + operator, if E1 is an array object (equivalently, a pointer to the initial element of an array object) and E2 is an integer, E1[E2] designates the E2-th element of E1 (counting from zero).

Therefore, **a[b]** is defined as : **a[b] == *(a + b)**

Similarly, **a[8] == *(a + 8)** Here, a is a pointer to the first element of the array and a[8] is the value of an elements which is 8 elements further from a, which is the same as *(a + 8) and **8[a]** will evaluate to following which means both are same.

8[a] == *(8 + a)

So by addition commutative property, **a[8] == 8[a]**

Multi-dimensional arrays

In C, we can define multidimensional arrays in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

General form of declaring N-dimensional arrays:
data_type array_name[size1][size2]....[sizeN];
data_type: Type of data to be stored in the array. Here data_type is valid C data type
array_name: Name of the array
size1, size2,... ,sizeN: Sizes of the dimensions

Size of multidimensional arrays

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

The array **int x[10][20]** can store total (10*20) = 200 elements.

Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

Initializing Three-Dimensional Array: Initialization in Three-Dimensional array is same as that of Two-dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

Method 1:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23};
```

Better Method:

```
int x[2][3][4] = { { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} }, {
{12,13,14,15}, {16,17,18,19}, {20,21,22,23} } };
```

Accessing elements in Three-Dimensional Arrays: Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

int arr [20][30][40];

Each element can be accessed as: arr [i][j][k]

Write a program to find the sum of elements in a 3-D array [2x3x4]

```
int numbers[2][3][4] = {
        { // First block of 3 rows
                { 10, 20, 30, 40 },
                { 15, 25, 35, 45 },
                { 47, 48, 49, 50 }
        },
        { // Second block of 3 rows
                { 10, 20, 30, 40 },
                { 15, 25, 35, 45 },
                { 47, 48, 49, 50 }
        }
};
```

Here’s how you could sum the elements:

```
int sum = 0;
for(int i = 0 ; i < 2 ; ++i)
{
        for(int j = 0 ; j < 3 ; ++j)
        {
                for(int k = 0 ; k < 4 ; ++k)
                {
                        sum += numbers[i][j][k];
                }
        }
}
printf("The sum of the values in the numbers array is %d.", sum);
```

How do you find size of every dimension in a multi-dimensional array?

```
#include <stdio.h>
int main (void)
{

    int array[11][22][33];
    int x1 = sizeof(array) / sizeof(*array);  /* = 11 */
    int x2 = sizeof(array) / sizeof(array[0]);  /* = 11 */
    int y1 = sizeof(*array) / sizeof(**array);  /* = 22 */
    int y2 = sizeof(array[0]) / sizeof(array[0][0]);  /* = 22 */
    int z1 = sizeof(**array) / sizeof(***array);  /* = 33 */
    int z2 = sizeof(array[0][0]) / sizeof(array[0][0][0]);  /* = 33 */

   printf ("%d  %d  %d\n", x1, y1, z1);
   printf ("%d  %d  %d\n", x2, y2, z2);
   return 0;
}
```

Multi-dimensional arrays and pointers

```
//  Multidimensional arrays and pointers
#include <stdio.h>
int main(void)
{
    char board[3][3] =
    {
        {'1','2','3'},
        {'4','5','6'},
        {'7','8','9'}
    };
    char *pboard = *board; // A pointer to char

    for(int i = 0 ; i < 9 ; ++i)
        printf(" board: %c\n", *(pboard + i));
    return 0;
}
```

What will be the output of this program?

How it works

You initialize pboard with the address of the first element of the array, and then you use normal pointer arithmetic to move through the array:

```
char *pboard = *board; // A pointer to char
for(int i = 0 ; i < 9 ; ++i)
    printf(" board: %c\n", *(pboard + i));
```

Note how you dereference board to obtain the address you want (with *board). This is necessary because board by itself is of type char**, a pointer to a pointer, and is the address of the subarray board[0]. It is not the address of an element, which must be of type char*.

You could have initialized pboard by using the following:
char *pboard = &board[0][0];
This amounts to the same thing. You might think you could initialize pboard using this statement:
pboard = board; // Wrong level of indirection!
This is wrong. You should at least get a compiler warning if you do this and ideally it should not compile at all.

Strictly speaking, this isn't legal because pboard and board have *different levels of indirection. That's a great* jargon phrase that just means that pboard refers to an address that contains a value of type char, whereas board refers to an address that *refers to an address containing a value of type char. There's an extra level with* board compared to pboard. Consequently, pboard needs one * to get to the value and board needs two **. Some compilers will allow you to get away with this and just give you a warning about what you've done.

However, it is an error, so you shouldn't do it!

Accessing Array Elements

Table (next slide) lists different ways of accessing your **board** array. The left column contains row index values to the **board** array, and the top row contains column index values. The entry in the table corresponding to a given row index and column index shows the various possible expressions for referring to that element.

Table - Pointer Expressions for Accessing Array Elements

board	0	1	2
0	board[0][0] *board[0] **board	board[0][1] *(board[0]+1) *(*board+1)	board[0][2] *(board[0]+2) *(*board+2)
1	board[1][0] *(board[0]+3) *board[1] *(*board+3)	board[1][1] *(board[0]+4 *(board[1]+1) *(*board+4)	board[1][2] *(board[0]+5) *(board[1]+2) *(*board+5)
2	board[2][0] *(board[0]+6) *(board[1]+3) *board[2] *(*board+6)	board[2][1] *(board[0]+7) *(board[1]+4) *(board[2]+1) *(*board+7)	board[2][2] *(board[0]+8) *(board[1]+5) *(board[2]+2) *(*board+8)

typedef

typedef is a reserved keyword in the C programming language. It is used to create an alias name for another data type. As such, it is often used to simplify the syntax of declaring complex data structures consisting of struct and union types, but is just as common in providing specific descriptive type names for integer data types of varying lengths.

```
typedef unsigned char USCH;
typedef short int_16;
```

```c
// C program to demonstrate typedef
#include <stdio.h>
// After this line BYTE can be used in place of
// unsigned char
typedef unsigned char BYTE;

int main(void)
{
    BYTE b1, b2;
    b1 = 'c';
    printf("%c ", b1);
    return 0;
}
```

```c
// C program to demonstrate #define
#include <stdio.h>

// After this line HYD is replaced by
// "Hyderabad"
#define HYD "Hyderabad"

int main(void)
{
    printf("%s ", HYD);
    return 0;
}
```

Difference between typedef and #define:

1. typedef is limited to giving symbolic names to types only, whereas #define can be used to define an alias for values as well, e.g., you can define 1 as ONE, 3.14 as PI, etc.
2. typedef interpretation is performed by the compiler where #define statements are performed by preprocessor.
3. #define should not be terminated with a semicolon, but typedef should be terminated with semicolon.
4. #define will just copy-paste the definition values at the point of use, while typedef is the actual definition of a new type.
5. typedef follows the scope rule which means if a new type is defined in a scope (inside a function), then the new type name will only be visible till the scope is there. In case of #define, when preprocessor encounters #define, it replaces all the occurrences, after that (No scope rule is followed).

Structure

What is a structure?

A structure is a user defined data type in C.

A structure creates a data type that can be used to group items of possibly different types into a single type.

How to create a structure?

'struct' keyword is used to create a structure.

```
struct address
{
   char name[50];
   char street[100];
   char city[50];
   char state[20];
   int pin;
};
```

How to declare structure variables?

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

How to initialize structure members?

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
   int x = 0;  // COMPILER ERROR:  cannot initialize members here
   int y = 0;  // COMPILER ERROR:  cannot initialize members here
}
```

Reason?

The reason for above error is simple, when a data type is declared, no memory is allocated for it. Memory is allocated only when variables are created.

```
struct individual
{
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
};

struct individual robbin= {
24, 17,“Robbin", “ABC", “XYZ"
};
```

The struct keyword is required when you define a new variable that stores a structure, but the code would look simpler and easier to read without it. There’s a way you can remove the need to include struct every time you declare a variable by using a typedef definition. For example:

typedef struct individual INDIVIDUAL;

This defines INDIVIDUALto be the equivalent of struct individual. If you put this definition at the beginning of a source file, you can define a variable of type INDIVIDUAL like this:

```
INDIVIDUAL rama= {
30, 15, “Rama", “Bhima", “Hidimbi"
};
```

The struct keyword is no longer necessary. This makes the code less cluttered and makes your structure type look like a first-class type.

Accessing Structure Members

You refer to a member of a structure by writing the variable name followed by a period, followed by the member variable name.

The period between the structure variable name and the member name is called the *member selection operator.*
Ex: rama.age = 40;

Structure members are the same as variables of the same type. You can set their values and use them in expressions in the same way as ordinary variables.

You have the option of specifying the member names in the initialization list, like this:

```
INDIVIDUAL trigger = { .height = 15, .age = 30,
.name = "Trigger", .mother = "Wesson", .father = "Smith"
};
```

Now there is no doubt about which member is being initialized by what value. **The order of the initializers is now unimportant**.

Pointer to a structure

```
INDIVIDUAL *pindividual;
```

How do you access the individual members of the structure?

```
pindividual->name
pindividual->age
pindividual->father
pindividual->mother
```

A simple program

```
#include <stdio.h>
typedef struct individual
{
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
} INDIVIDUAL;

void get_values (INDIVIDUAL *per)
{
    printf ("Key in the name:");
    scanf ("%s", per->name);
    printf ("Key in the age:");
    scanf ("%d", &per->age);
}
```

```
void display_values (INDIVIDUAL per)
{
    printf ("Name is %s\n", per.name);
    printf ("Age is %d\n", per.age);
}

int main (void)
{
    INDIVIDUAL per;
    get_values (&per);
    display_values (per);
    return 0;
}
```

Operations on structures

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with &, and accessing its members. Copy and assignment include passing arguments to functions and returning values from functions as well. Structures may not be compared. A structure may be initialized by a list of constant member values; an automatic structure may also be initialized by an assignment.

Unnamed Structures

You don’t have to give a structure a tag name. When you declare a structure and any instances of that structure in a single statement, you can omit the tag name. In the previous example, instead of the structure declaration for type INDIVIDUAL, followed by the instance declaration for my_brother, you could have written this statement:

```
struct
{ // Structure declaration and...
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
} my_brother; // ...structure variable declaration combined
```

A serious disadvantage with this is that you can no longer define further instances of the structure in another statement. All the variables of this structure type that you want in your program must be defined in the one statement.

Arrays of structures

Write a program to accept the details of five individuals and print out the same.

Size of a structure

It's very important to use sizeof when you need the number of bytes occupied by a structure. it doesn't necessarily correspond to the sum of the bytes occupied by each of its individual members, so you may get it wrong if you try to work it out yourself. **Variables other than type char are often stored beginning at an address that's a multiple of two for 2-byte variables, a multiple of four for 4-byte variables, and so on. this is called *boundary alignment and it has nothing to do with C in particular but it can be a hardware requirement*.**

Arranging variables to be stored in memory like this makes the transfer of data between the processor and memory faster. This arrangement can result in unused bytes occurring between member variables of different types, though, depending on their sequence. these have to be accounted for in the number of bytes allocated for a structure.

A quick recap

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called "records" in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

One traditional example of a structure is the payroll record: an employee is described by a set of attributes such as name, address, social security number, salary, etc. Some of these in turn could be structures: a name has several components, as does an address and even a salary. Another example, more typical for C, comes from graphics: a point is a pair of coordinates, a rectangle is a pair of points, and so on.

Structures can be nested;

```
struct point
{
    int x;
    int y;
};
```

One representation of a rectangle is a pair of points that denote the diagonally opposite corners

```
struct rect
{
    struct point pt1;
    struct point pt2;
};
```