FIT2100 Tutorial #2
More on C Programming: Pointers and Dynamic Structures
Week 3 Semester 2 2019

August 7, 2019

Revision Status:

Written by Jojo Wong, 2018.

Updated by Daniel Kos, Aug 2019.

Acknowledgement:
The majority of the content presented in this tutorial was adapted from the courseware of FIT3042 prepared by Robyn McNamara.

Contents

1 Background

In this second tutorial, we will continue to explore the C programming language.

We will first discuss the various control flow structures used in C; followed by the advanced data structures supported by C, which include the pointers, the user-defined structures (`struct`), as well as the dynamic structures.

You should complete the following reading (Section 2) before attending the tutorial. You should also prepare the solutions for the practice tasks given in Section 3.

2 Pre-tutorial Reading (2 marks)

2.1 Statements and Blocks

Statements Any valid expression terminated with a semicolon (;) is regarded as a statement in C.

```
1  x = y + z;
2  scanf("%d", &number);
3  printf("%s", name);
```

Statement blocks Multiple statements (including declarations) can be grouped together using braces ({}) to form compound statements or statement blocks. We have seen that the statements of a function (i.e. the function body) are surrounded by a pair of braces.

Statement blocks are used in various control flow structures in C which determine the order in which statements are executed when certain conditions are given.

2.2 Selective Structures

2.2.1 If and If-Else Statements

Both `if` and `if-else` statements are supported in C with the following syntax:

```
if (x == y)
{
    printf("equal");
}
```

```
if (x == y)
{
    printf("equal");
}
else
{
    printf("not equal");
}
```

```
if (x < y)
{
    if (x < z)
        min = x;
    else
        min = z;
}
else
{
    if (y < z)
        min = y;
    else
        min = z;
}
```

Note: The parentheses around the expression (condition) are required; unlike in other programming languages (such as Python) which can be omitted.

Alternatively, the condition expression can be used as a "shorthand" for non-nested `if-else` statements.

```
printf("%s", (x == y) ? "equal" : "not equal");
```

2.2.2 Switch Statement

C provides the `switch` statement to handle multiple decisions based around conditions expressed as integer values (`int`) or character constants (`char`).

```c
/* day is defined as an integer int */

switch (day)
{
    case 1:
        printf("MONDAY");
        break;
    case 2:
        printf("TUESDAY");
        break;
    case 3:
        printf("WEDNESDAY");
        break;
    case 4:
        printf("THURSDAY");
        break;
    case 5:
        printf("FRIDAY");
        break;
    case 6:
        printf("SATURDAY");
        break;
    case 7:
        printf("SUNDAY");
        break;
    default:
        printf("unknown");    /* for integers not in range of 1-7 */
        break;
}
```

Note: A `break` statement is required at the end of the statement block for each case, which will cause an immediate exit from the `switch` statement.

2.3 Iterative Structures

2.3.1 While and Do-While Loops

Both `while` and `do-while` loops are available in C.

```c
while (condition)
{
    /* statements */
}
```

```
do
{
    /* statements */
} while (condition);    /* a semicolon is required here */
```

The test condition (expression) in a `while` loop is evaluated before each iteration; whereas for a `do-while` loop, the test condition is evaluated at the end of each iteration.

2.3.2 For Loop

The `for` loops in C have the very similar structures in Java. The header of a `for` loop consists of three components: the *initialisation*, the *test condition*, and the *increment or decrement step*.

You can omit any of the three components provided the semicolons are retained. However, omitting the condition will result in a "infinite" loop (since it is considered as a permanent `true` condition).

```
for (initialisation; condition; step)
{
    /* statements */
}
```

```
#define MAX 5

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

for (int i = 0; i < MAX;  i += 2)
{
    printf("%d\n",  numbers[i]);      /* 1, 3, 5, 7 and 9 are printed */
}
```

Note: C does not support the convenient way for iterating through the elements of a sequence or a collection (such as the `for-each` loop in Java or the `for-in` loop in Python).

2.4 Pointers in C

Pointers store addresses in memory. A pointer may contain the address of another variable (as a "reference" to another value). When assigning the address of a variable to a pointer, the pointer is said to "point to" that variable.

Syntactically, C users the asterisk (*) to indicate or declare a pointer. The pointer type is defined based on the type of the variable that it points to. In the example below, `iptr` is a pointer which refers to a variable of type `int`.

The & operator is used to set a pointer to point to specific variable, by assigning the address of that variable to the pointer. To obtain the value of the variable that a pointer refers to (i.e. de-referencing), the * operator is used.

```
int *iptr;              /* iptr is a pointer to int */
int x = 3;

iptr = &x;              /* iptr points to x */
printf("%d\n", *iptr);  /* 3 is printed */
```

Note: A pointer can only be assigned to the type of the variable that it points to (with one exception for the pointer type of `void *`). A pointer can also be assigned with the special value `NULL` when it is not pointing at anything.

Changing pointers When changing the value of the variable that a pointer (`iptr`) is pointing at, that also changes the value of `*ptr`; but not the pointer itself (i.e. it is still having the same address of the variable that it points to).

```
++x;                    /* x = 4 and *iptr = 4 */
++(*iptr);              /* x = 5 and *iptr = 5 */
```

To change what a pointer points to, we can simple re-assign it with a different variable (or a new memory address) to it.

```
int y = 10;
iptr = &y;              /* iptr points to y now */
printf("%d\n", *iptr);  /* 10 is printed */
```

Two (or more) pointers can be made to refer to the same variable (i.e. the value of each of these pointers is the same which is the address of that variable.)

```
int *iptr1, *iptr2;       /* iptr1 and iptr2 are pointers to int */
int x = 3;

iptr1 = &x;               /* iptr1 points to x */
iptr2 = iptr1;            /* iptr2 points to where iptr1 is pointing */

printf("%d\n", *iptr1);   /* 3 is printed */
printf("%d\n", *iptr2);   /* 3 is printed */
```

Void pointers The special type of pointer with the `void` type allows *conversion* between different types. However, it must be used with care since no type checking is performed.

```
int *iptr;                  /* iptr is a pointer to an integer*/
double *dptr;               /* dptr is a pointer to a double */
void *vptr;                 /* vptr is a pointer to any type */

iptr = dptr;                /* illegal */
vptr = (void *) iptr;       /* legal */
dptr = (double *) vptr;     /* legal */
```

2.4.1 Pointers and Function Arguments

In C, functions arguments are always *passed by value*. With this, it is impossible for the called function to alter the original variables in the calling function.

The approach known as *passed by reference* can be used to pass pointer arguments to the values to be modified in the called function. As we have seen, pointers refer to the actual addresses of the variables – hence, a pointer can directly access the variable that it points to and modify the content of that variable.

```
/* passing arguments by value */

void swap1(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}

/* passing arguments by reference */

void swap2(int *xptr, int *yptr)
{
  int temp;

  temp = *xptr;
  *xptr = *yptr;
  *yptr = temp;
}

```

```
23  int a = 1, b = 2;
24  swap1(a, b);                /* after swap1 was executed, a = 1, b = 2 */
25  swap2(&a, &b);              /* after swap2 was executed, a = 2, b = 1 */
```

2.4.2 Pointers and Arrays

Recall that, as like other programming languages (such as Python and Java), C uses the `[]` operator to index arrays. This is in fact a "shorthand" of using pointers.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int i;    /* counter used in loops */
6
7       /* create an array initalised with 5 elements */
8       int array[5] = {11, 3, 6, -1, 8};
9
10      /* access array using subscript notation */
11      for (i = 0; i < 5; i++)
12          printf("a[%d] = %d\n", i, array[i]);
13
14      /* access array using pointer notation */
15      for (i = 0; i < 5; i++)
16          printf("*(a + %d) = %d\n", i, *(array + i));
17
18      return 0;
19  }
```

To refer to the `i`-th element of an array, `a[i]` is equivalent to `*(a + i)`, since the name of an array can be regarded as a pointer (i.e. indicating the starting address of the array and also its first element).

Note: Arithmetic operations — addition and subtraction — can be applied on pointers just like any other basic data types.

2.4.3 Pointers and Strings

Recall that C has no built-in string type like Java and Python. In C, a string is in fact an array of `chars` terminated by a '`\0`' (`null`) character. Likewise, we could also define a string using a pointer.

```c
char array_str[] = "FIT2100";      /* an array of characters */
char *ptr_str = "FIT2100";         /* a pointer to a string literal */

printf("%s\n", array_str);
printf("%s\n", ptr_str);
```

The <string.h> library contains many common functions for string manipulation. Many of these functions often manipulate strings using pointers of type (char *).

It is the *programmers' responsibility* to ensure sufficient memory is allocated and available for storing whatever the strings will need to store, given that the string functions generally do not perform checking on the memory space.

Note: Each string must be terminated with '\0'; otherwise the string functions may fail (and your program would crash unexpectedly).

String Function	Description
char * strcpy(char *dst, char *src)	copy from src to dst
char * strcat(char *str, char *new)	append the string new to str
char * strchr(char *str, int chr)	locate the character chr in str
int strcmp(char *str1, char *str2)	compare str1 and str2
size_t strlen(char *str)	return the length of str (but not including '\0')

The command-line arguments We have seen that the main function often accepts two arguments from the command line: int argc and char *argv[]. The second argument is in fact an array of character strings or char pointers of variable length.

Given that argv is a pointer to an array of pointers, we can access each of the argument strings by using the pointer rather than the array index.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    while (--argc > 0)
    {
        printf("%s%s", *(++argv), (argc > 1) ? " ": "");
    }
    printf("\n");
    return 0;
}
```

Note: (++argv) causes the pointer to point at argv[1] instead of argv[0] in the first iteration, since the first argument is in fact the program (command) name.

2.5 Structs in C

Often times, if we need to perform complex computation, we will have to define our own new data structures to make your program more organised and manageable.

We have seen that data of the same type can be organised as arrays. To handle complicated data, C provides a data structure known as `struct` (structures) to group data of different types together under a single name for easy handling.

Structs in C are different to classes in Python or Java with respect to the following:

- No methods are defined in structs;
- Differences in declaration syntax;
- The arrow (->) operator is used for accessing through pointers;
- The `new` operator is not used during instantiation.

Declaring a struct The keyword `struct` is used for declaring a new structure. The structure tag (following the keyword `struct`) is optional; it can however be used as a "shorthand" when creating variables based on this structure.

```
1  struct Point
2  {
3    int x;
4    int y;
5  };
```

Using a struct To create variables of a specific `struct` type, we can do the following if the structure tag is given during declaration.

Each member (field) defined in a `struct` for each variable is accessed using the dot (.) notation. To access members of a struct via pointers, the array (->) operator is used.

```
1  struct Point point1, point2;  /* struct keyword must be included */
2
3  point1.x = 0;
4  point1.y = 0;
5
```

```
6   point2.x = 1;
7   point2.y = 1;
8
9   printf("%d %d\n", point1.x, point1.y);  /* point1.x = 0; point1.y = 0 */
10  printf("%d %d\n", point2.x, point2.y);  /* point2.x = 1; point2.y = 1 */
```

```
1   struct Point point1;
2   struct Point *ptr;    /* ptr is a pointer to struct Point */
3
4   point1.x = 0;
5   point1.y = 0;
6
7   ptr = &point1;
8
9   printf("%d %d\n", point1.x, point1.y);  /* point1.x = 0; point1.y = 0 */
10  printf("%d %d\n", ptr->x, ptr->y);      /* ptr->x = 0; ptr->y = 0 */
```

Define a struct type More conveniently, we can define our own type name using the keyword `typedef`. The name of the `struct` type must come at the end, but not after the `struct` keyword.

```
1   typedef struct
2   {
3     int x;
4     int y;
5   } Point;    /* a semicolon is required here */
6
7   Point point1, point2;
```

Note: We can also give *aliases* to built-in data types by using `typedef`. By performing this does not actually make the C compiler to produce different code; however it can make your code more readable. (The common practice is to include `typedef` statements in the `.h` header file.)

```
1   typedef unsigned long int size_t;
```

2.6 Dynamic Data Structures

Suppose that you have declared a `struct` of type `Name` which contains pointers:

```c
typedef struct
{
  char *first_name;
  char *middle_name;
  char *last_name;
} Name;
```

By just declaring an instance (creating a variable) of `Name`, does not mean that memory space is (automatically) allocated to store each of the three strings (`char` pointers). You will have to manually request memory space from the operating system (OS).

2.6.1 Memory Allocation and Deallocation

Dynamic memory allocation The `malloc(size)` function from the `<stdlib.h>` library allocates the number of bytes of memory space based on the value of `size` and returns a pointer to the allocated memory block (from the "heap" segment of the system memory).

```c
#include <stdlib.h>

/* function prototype */
void *malloc(size_t size);

/* allocate space for 10 intergers */
int *iptr = (int *) malloc (sizeof(int) * 10);

/* allocate space for a string of 20 characters including '\0' */
char *first_name = (char *) malloc (sizeof(char) * 20 + 1);
```

If the memory allocation is unsuccessful (e.g. out of memory), a NULL value is returned by `malloc` indicating that the memory request can be not satisfied. Otherwise, `malloc` returns a `void *` pointer, which must be cast to an appropriate type.

Note: The `size_t` is essentially an `unsigned long` integer which indicated the size of a memory block requested in terms of *bytes*.

Releasing dynamic memory Any memory dynamically allocated during run-time must be released (de-allocated) and returned back to the heap by using the `free(ptr)` function (from `<stdlib.h>`), where `ptr` refers to the memory block to be de-allocated.

```c
#include <stdlib.h>

/* function prototype */
```

```
4   void free(void *ptr);
5
6   /* de-allocate memory pointed by iptr */
7   free(iptr);
```

Programs that fail to free up dynamic memory may suffer from "memory leaks" which may consume the entire system's resources; and result in system slow-down and prevents other programs from using the unused memory.

Note: You should not attempt to de-reference a `NULL` or freed pointer. Attempting to access a de-allocated memory block will cause undefined behaviour that may result in a program crash.

2.6.2 Multi-dimensional Arrays

Pointers and dynamic memory can be used to construct multi-dimensional arrays of arbitrary size.

```
1   typedef int DataType;
2
3   DataType array[][];    /* an array of arrays */
4   DataType **array;      /* a pointer to a pointer */
```

To allocate memory for a multi-dimensional array, the outer dimension should first be allocated with the memory. This is then followed by filling the inner dimensions.

```
1   DataType ** create_2Darray (int dimx, int dimy, DataType initital_value)
2   {
3       int i, j;   /* counters */
4       DataType **array;
5
6       /* allocate an array of pointers */
7       array = (DataType **) malloc (sizeof(DataType *) * dimy);
8       assert(array);
9
10      for (i = 0; i < dimy; i++)
11      {
12          /* allocate an array of DataTypes */
13          array[i] = (DataType *) malloc (sizeof(DataType) * dimx);
14          assert(array[i]);
15
16          for (j = 0; j < dimx; j++)
17            array[i][j] = initial_value;
18      }
```

```
19      return array;
20  }
```

As for the de-allocation process, memory is released in reverse order.

```
1   void free_2Darray (DataType **array, int dimy)
2   {
3       int i;
4
5       for (i = 0; i < dimy; i++)
6       {
7         free(array[i]);    /* free the memory of each row */
8       }
9       free(array);
10  }
```

More on arrays and pointers

What is the difference between the following two declarations?

```
1   int arr1[10][15];
2   int *arr2[10];
```

To access an element, `arr1[2][3]` and `arr2[2][3]` are both syntactically legal statements.

However, `arr1` is a true 2-dimensional array with 150 `int`-sized memory locations have been set aside (10 rows x 15 columns). Unlike `arr2` which is an array of pointers, memory space is only allocated to the 10 `(int *)` pointers without initialisation.

Memory must be explicitly allocated to each of the rows in `arr2`. (Note that each row in `arr2` may be of variable length, i.e. each row of `arr2` does not have to point to 15 elements.)

In general, arrays of pointers are useful in particularly when you need arrays of pointers to objects of different length.

```
1   /* length of each argument string is not known at compile-time */
2   char *argv[];
3
```

```
4  /* represent a collection of strings of variable length */
5  char *month[] = {"Illegal month", "January", ... , "December"};
```

2.7 Debugging C programs

Some kinds of errors, such as typos and incorrect algorithms, are equally common in all programming languages. However, there are some kinds of errors that are more likely to affect C programs, such as buffer overflows and misuse of pointers.

The underlying reason is that C generally lacks safeguards that other programming languages may have. C does not attempt to hide implementation details from the programmers; likewise C does not prevent the programmers from attempting to access memory that is not allocated to the programmers' code. Notice that one of the common error under Unix/LInux is "segmentation fault" (`segfault`).

What is a segmentation fault? A segmentation fault occurs when you attempt to access memory that has not been allocated to it. It is generally a signal sent from the operating system (OS) to your program, and causing your program to exit. The OS that manages memory will not let your program to read from or write to memory segments that do not own by your program.

2.7.1 Handling errors with defensive programming

The best approach to handle errors in C is by preventing errors through "defensive programming" techniques.

- Always check return values from functions (such as `malloc()` and `fopen()`)
- Always initialise pointers to `null` on creation if the valid target is not given immediately
- Always check that pointers are not `null` before deferencing them (see below)

```
1  if (ptr_str) { //check not a null pointer
2    *ptr_str = "FIT2100";
3  }
```

Logging and diagnostics Often times, defensive programming is unable to catch all classes of errors. You can however instrument your code by getting it to output informative messages. For instance, `fprintf(stderr)` is one way to handle this[1] — `fprintf()` can also be used to output messages to a logfile. Alternatively, use `prinf()` as diagnostic statements in your code.

Note that these programming techniques are not specific to the C language or Unix/Linux. You would have certainly used diagnostic output statements in any other programming language that you have programmed in.

2.7.2 The `gdb` debugger

A debugger is a program that runs your program inside it, which allows you to inspect what the program is doing at a certain point during its execution. The debugger also enables you to manipulate the program state while it is running. As such, errors such as segmentation faults may be easier to detect with the help of a debugger.

A debugger that is often used with the C compiler gcc under the Unix/Linux environment is gdb — the GNU Debugger.

Usually, you would compile a C program as follows:

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

$ gcc sourcefile1.c sourcefile2.c ... -o myprogram
```

To run your program with the gdb debugger, you add a -g option to enable the built-in debugging support which is needed by the gdb, as with `valgrind`.

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

$ gcc -g sourcefile1.c sourcefile2.c ... -o myprogram
```

Starting up the debugger To start up the gdb debugger, you could just try with one of the following commands. (Note: `myprogram` is the executable program that you would like to debug.)

```
$ gdb
```

[1] This sends output to the program's 'standard error' stream; it will be displayed on the terminal even if the program's standard output has been redirected.

```
$ gdb myprogram
```

You will then get a prompt that looks as follows:

```
(gdb)
```

If you didn't specify a program to debug while starting up the debugger, you will have to load it with the "`file`" command after started up the debugger.

```
$ gdb
(gdb) file myprogram
```

Running your program with the debugger To run your program (say `myprogram`), just type "`run`":

```
(gdb) run
```

Your program will get executed. There are two possible outcomes here:

- If there are no serious problems (such as, no segmentation fault occurred), your program should run accordingly with the program output presented.
- If your program did have some problems, you should be getting some useful information about the error. (See below for an example of the error message.)

```
Program received signal SIGSEGV, Segmentation fault.
0x00007fff916e4152 in strlen () from /usr/lib/system/libsystem_c.dylib
```

The "`where`" command can be used to find out the line of code where your program crashed. This is in fact the metadata that the -g flag includes in the executable program.

Another command that can be helpful for finding out where your program crashed is the "`backtrace`" command — which produces a stack trace of the function calls that lead to a segmentation fault. (An example of a stack trace is shown below.)

```
(gdb) backtrace
#0  0x00007fff916e4152 in strlen () from /usr/lib/system/libsystem_c.dylib
#1  0x00007fff91729a54 in __vfprintf () from /usr/lib/system/libsystem_c.dylib
#2  0x00007fff917526c9 in __v2printf () from /usr/lib/system/libsystem_c.dylib
#3  0x00007fff9172838a in vfprintf_l () from /usr/lib/system/libsystem_c.dylib
#4  0x00007fff91726224 in printf () from /usr/lib/system/libsystem_c.dylib
#5  0x0000000100000f2c in main () at test.c:13
```

2.7.3 Dealing with the program bugs

If your program is running successfully, you would not need a debugger like gdb. But what if your program is not running properly or crashed? If that is the case, you do not want to run your program without stopping or breaking at a certain point during the execution. Rather, what you would want to do is to *step through* your code a bit at a time, until you discovered where the error is.

Setting the breakpoints A *breakpoint* instructs the gdb debugger to stop running your program at a designated point. To set breakpoints, use the command "`break`" in one of the following ways:

- To break at line 5 of the current program:

```
(gdb) break 5
```

- To break at line 5 of a source file named `sourcefile.c`:

```
(gdb) break sourcefile.c:5
```

- To break at a function named `myfunction.c`:

```
(gdb) break myfunction
```

You can set multiple breakpoints within your program. If your program reaches any of these locations when running, your program will stop execution and prompt you for another debugger command.

Once you have set a breakpoint, you can try to run your program with the command "`run`". The program should then stop where you have instructed the debugger to stop — provided no fatal errors occurred before reaching the breakpoint.

To proceed with the next breakpoint, use the "`continue`" command. (You should not try to use the "`run`" command here as that would restart the program from the beginning, which is not going to be helpful.)

Note: Any breakpoints can be removed with the command "`clear`" — e.g. to delete the breakpoint at line 5:

```
(gdb) clear 5
```

Tracing your code When you program hits the breakpoint, you can step through your program one line at a time with the following commands:

- "`step`" command: go to the next line, stepping *into* any functions that the current line calls
- "`next`" command: go to the next line, but stepping *over* any functions that the current line calls
- "`finish`" command: finish the current function and *break* when it returns

Keeping an eye on variables You can also tell the gdb debugger to show the values of various variables or expressions at all times using the "`display`" or "`print`" command. Both of these commands take an optional format argument to display the value in a specific format:

- "`print/x`" command: print in hexadecimal
- "`print/c`" command: print in character
- "`print/t`" command: print in binary

```
(gdb) print myvariable
(gdb) print/x myvariable
```

If you would prefer to monitor or watch a particular variable whenever its value changes, use the "`watch`" command. The program will be interrupted whenever a watched variable's value is modified.

```
(gdb) watch myvariable
```

If you ever get confused about any of the gdb commands, use the "`help`" command with or without an argument.

```
(gdb) help [command]
```

Finally, to exit the gdb debugger is the "`quit`" command.

```
(gdb) quit
```

2.7.4 Pre-class exercise (2 marks)

For the given C program below, type it out and name the source file as "`test.c`". Then, run through it with the `gdb` debugger. You will need to step through the program code to detect where the error occurred. Once you have found the error, fix it so that the program will run successfully.

```c
#include <stdio.h>

int main(void)
{
        char str1[] = "FIT2100 Operating Systems";
        char str2[] = "Week 3 Tutorial";
        char *ptr = NULL;

        ptr = str1;
        printf("%s\n", ptr);

        ptr = str2;
        printf("%s\n", *ptr);

        return 0;
}
```

3 Practice Tasks

3.1 Task 1

Write a C program that converts the 24-hour time format into the 12-hour time format. The program should first prompt for a time in the 24-hour format, and then display the equivalent 12-hour format. For instance, if the input is `13:15`, the program should print `1:15 PM`.

3.2 Task 2

Write a C program that reads a two-digit number and presents the number in English words. The program should make use of `switch` statements to select the corresponding English word associated with each digit of the number. For instance, the input of 68 should be printed as "sixty eight" .

3.3 Task 3

Write a C program that computes the *greatest common divisor* (GCD) between two positive integers. For instance, the GCD for integers 12 and 78 is 6. The program should implement the Euclidean algorithm to find the GCD.

3.4 Task 4

Write a function in C that accepts an array of integers and the length of the array as arguments. The function should then find the smallest element in the given array by returning a *pointer* to the position of the smallest element. The function prototype is given as follows:

```
1  int *find_smallest(int array[], int n);
```

3.5 Task 5

Given the following `struct` type called `Name`:

```
1  typedef struct
2  {
3    char *first_name;
4    char *last_name;
5  } Name;
```

Create two variables of type `Name` and initialise them with some appropriate values. Write a function in C that determines whether these two `Name` variables are equal to each other.

(Note: The program should perform comparison on both the `first_name` and `last_name` to check for *equality*. You may use the `strcmp` function from the `<string.h>` library.)