THE

C

PROGRAMMING

LANGUAGE

Collected

From

www.sikkhok.com

By-Engr Rezaul Karim

USTC,FSET.

C Basics: Introduction

This module discusses C, which is undoubtedly one of the most popular programming languages. Several languages are based on C. C++ extends C by adding object-oriented paradigm, among others. Java uses a similar syntax as that of C. CPython -- one of the popular versions of Python -- is written on top of C.

This module begins with a coverage of basic C concepts like variables, macros, arrays, strings, enums, loops, conditional statements, etc., and then moves on to advanced concepts like pointers, data-structures, linked-lists, socket-programming, POSIX threads, and many more! We supplement all of these discussions with a lot of code examples to facilitate understanding.

Elements of a C program

Stated simply, a C program is essentially a collection of statements that follow of rules (syntax). Let us dive right into it and take a look at a simple C program -- we use this program to understand some of the common syntax element. So, here is the program:

```
#include <stdio.h>

int main () {
    /* This will print "Hello World". */
    printf("Hello World\n");

    return 0;
}
```

The above program begins by including a header file ("stdio.h") -- "stdio" is short for standard input output (IO). Header files, like "stdio.h", include definitions of functions, data structures, constant values, etc. For example, one of the function definitions provided by "stdio.h" is that of the printf() call, which is used for printing values on the standard output. If we were to call printf() function without including the "stdio.h", the compiler would complain that it can not find definition of printf() function. We will revisit functions and printf() later.

Next, every C-program needs to have a main() function. When we run the program, the program runs this function before any other function. A C program would typically have a (whole) lot of functions and all of these functions are called directly or indirectly by the main() function; the latter means that function gets called by a function that gets called by the main function. You get the idea! And if we do not have the main() function, then the compiler would be certain to complain.

Moving on, we notice the line that contains the printf() function; the reason why we included "stdio.h"! C programs contains several such lines, each of them are called statements.

The next thing to note in the above program is that the line containing printf(), ends with a semicolon. A semicolon tells the compiler that it has reached the end of the current statement and thus, is different from the next statement (or the previous) statement. Using a semicolon to

terminate a statement is a popular style that is used in other languages as well: C++, Java, PHP, JavaScript, etc. However, in JavaScript using semicolon to terminate a statement is optional! So, if you have a JavaScript background, now would be a good time to take a note.

C supports additional types of statements as well. Some statements assign values to variables. Some statements can take input from the user. C also has control statements to provide control-logic to programs: conditional clauses like "if-else", looping clauses like "while", "for" etc. Please note that control statements typically do not require a semicolon at the end. These statements are the building blocks of any C program.

Further, the program contains text enclosed between "/*" and "*/" -- this is a comment and the compiler ignores it. Thus, using these opening and closing tags, we can add helpful notes explaining the program logic. In real life, it is not uncommon from programmers to skip adding comments; lack of comments can make code less readable for future programmers who need to maintain or extend it. Trust me, adding good comments would help you earn a lot of karma points!

In the end, the program contains a "return 0" statement -- this statement returns 0 to the caller of this program (e.g. the shell from where we run this program). In C, each function can return a value and the main() being a function itself, can also do the same. A return value of 0 from the main() function means that the program was run successfully.

Compiling and Running a C program

Once we have written the program and saved it to a file (let us call it "hello.c"), the next step is to compile it using a C compiler; it is this step that converts a simple text file storing the program into an executable that we can run. C differs from scripting languages like Python, Perl, JavaScript, where no compilation is necessary -- for those languages, the programs are checked for syntax errors during the run-time itself.

Compilation is a necessary step since if the program has any syntax errors (like, no semicolon at the end of a statement), then the compiler would tell us that. And the compiler would allow us to go only after we fix all of those errors. For beginner programmers, it is common to have frequent nasty fights with the compiler; if we run into a compilation error, we should just revisit the syntax rules. If we still face the issue, then we should try to google the error -- it is almost guaranteed that some poor soul must have run into the same issue! We should take comfort that as we write more code, spotting syntax errors starts to come naturally.

For our example, we use gcc compiler -- gcc stands for GNU compiler collection and contains compiler for various languages. To be more precise, we use gcc version 4.6.3 on Linux for this example and for all examples in this module. Besides gcc, there are other compilers as well (like Unix cc or Borland's Turbo cc (tcc)).

We have to keep in mind the operating system where we wish to compile C programs. If we are using systems other than Linux, then we might need the relevant compiler or a compiler bundle; for example, for Mac OSX systems, we need to install Xcode Developer Tools that contains the

gcc compiler. If we are using Windows, then we can use Borland's Turbo C. Note that it is likely that these compilers may be already installed on the system; if not, then we need to do the hard-work and install them.

When we compile a program, the output is stored in "a.out" by default. If we want, we can use the "-o" option to store the output in another location ("hello_o"). Once we have compiled, we can run the executable, "hello_o".

```
$ gcc --version
gcc (GCC) 4.6.3 20120306 (Red Hat 4.6.3-2)
Copyright (C) 2011 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
$
$ ls -alh hello.c
-rw-rw-r--. 1 user  user 105 Jan 12 22:12 hello.c
$
$ gcc hello.c -o hello_o
$
$ ./hello_o
Hello World
```

C Basics: Variable Types

Variables allow us to store data of different types. It is fairly easy to define and use a C variable. Let us see this using an example. The example (provided below) defines four variables: var_num1, var_num2, var_sum, and var_product. Next, it places the sum and product the first two variables (var_num1 and var_num2) into the last two variables (var_sum and var_product).

```
#include <stdio.h>

int main () {
    int var_num1 = 10;
    int var_num2 = 5;
    int var_sum, var_product;

    /* Store the sum of var_num1 and var_num2 in var_sum */
    var_sum = var_num1 + var_num2;

    /* Store the product of var_num1 and var_num2 in var_product */
    var_product = var_num1 * var_num2;

    printf("Variables: var_num1: %d, var_num2: %d\n", var_num1, var_num2);
    printf("var_sum: %d, var_product: %d\n", var_sum, var_product);
    return 0;
}
```

Let us compile and run the program.

```
$ gcc variable.c -o var_o
```

```
$ ./var_o
Variables: var_num1: 10, var_num2: 5
var_sum: 15, var_product: 50
```

The above program shows how to assign a value to a variable, "var_num1 = 10", or "var_sum = var_num1 + var_num2"; these statements show that the value on the right hand side, e.g. 10 gets assigned to the variable, var_num1. For assignment operations, the order is important -- assigning in the opposite way (e.g. "10 = var_num1") would be a compilation error. The variable on the left hand side is also sometimes known as an lvalue.

Depending upon where we define a variable, the scope of the variable is qualified. For example, if we define a variable outside of any function (including main()), then it can be used throughput the file. On the other hand, if we define a variable inside a function, then it can be used only inside that function. We will revisit both functions and variable scope later!

Storage Types for Variables

In the above program, both var_num1 and var_num2 store an integer (we provide the "int" qualifier before the variable names); an integer is a number (digit) without any fractions. C provides storage for several types of numbers. For example, if we were to store "10.50" as value of var_num1, then we could not do that since "10.50" is a fractional number and not an integer (try assigning "10.50" to var_num1 in the above program and recompiling it!). For such cases, C provides two additional types: float and double. Both int and float usually require a size of 4 bytes, whereas a double requires a size of 8 bytes.

Besides int type, there are two other types for storing integers in C: short and char. Variables of type "short" require 2 bytes and variables of type "char" require only 1 byte.

A char (or character) type can also store values of alphabets or other non-numeric characters. For example, we can use a char to store values like 'A' (which has an ASCII value of 50). Since a char variable uses only 1 byte (which is 8 bits), the maximum value of a digit that it can store is 255 ((2 to the power 8) - 1). Technically speaking, when we assign a character to a variable, then the variable actually stores the numeric ASCII value.

All of the above variables can store both positive and negative values. To do this, these variables use the leftmost bit to store if it is positive or negative. Thus, to store the sign of a value, we lose one bit of storage. To avoid this inefficient behavior, when working with only positive numbers, we can use an "unsigned" versions of char, short, and int: "unsigned char", "unsigned short", and "unsigned int". Due to reasons of efficiency, C does not support unsigned variants for float and double!

With that, let us use a small example to investigate variables further. This example defines variables of different storage types and prints their values/sizes; for that, it uses the built-in sizeof() function to retrieve size of a variable. Please note that the printf() function uses a "%f" for printing fractional numbers (float and double), "%d" for printing integer numbers, and "%u" for printing unsigned integers. We will revisit printf() a little later.

```
#include <stdio.h>

int main () {
    char var_char = -10;
    short var_short = -10;
    int var_int = -10;
    float var_float = -10.50;
    double var_double = -10.50;

    /* These are the unsigned variants */
    unsigned char var_uchar = 10;
    unsigned short var_ushort = 10;
    unsigned int var_uint = 10;

    printf("  var_char: %-5d (sizeof char: %d)\n", var_char,
sizeof(var_char));
    printf(" var_short: %-5d (sizeof short: %d)\n", var_short,
sizeof(var_short));
    printf("   var_int: %-5d (sizeof int: %d)\n", var_int, sizeof(var_int));
    printf(" var_float: %.1f (sizeof float: %d)\n", var_float,
sizeof(var_float));
    printf("var_double: %.1f (sizeof double: %d)\n", var_double,
sizeof(var_double));

    printf(" var_uchar: %-5u (sizeof unsigned char: %d)\n", var_uchar,
sizeof(var_uchar));
    printf("var_ushort: %-5u (sizeof unsigned short: %d)\n", var_ushort,
sizeof(var_ushort));
    printf("  var_uint: %-5u (sizeof unsigned int: %d)\n", var_uint,
sizeof(var_uint));
    return 0;
}
```

Assuming that the file name of above program is "sizes.c", here is its compilation and output:

```
$ gcc sizes.c -o size
$
$ ./size
  var_char: -10   (sizeof char: 1)
 var_short: -10   (sizeof short: 2)
   var_int: -10   (sizeof int: 4)
 var_float: -10.5 (sizeof float: 4)
var_double: -10.5 (sizeof double: 8)
 var_uchar: 10    (sizeof unsigned char: 1)
var_ushort: 10    (sizeof unsigned short: 2)
  var_uint: 10    (sizeof unsigned int: 4)
```

If we were to assign "-10" to var_uchar or any of the unsigned variables, then the value printed may not be "-10" because storing a negative number into an unsigned variable is undefined. Clearly, it is not a good idea to assign a negative value to an unsigned variable!

Variable Operators

Besides simple operators to do addition and multiplication, C provides additional arithmetic operators as well: subtraction using "-", division using "/", modulo using "%" etc. We should note that the modulo operator has an important constraint: both of the variables should be integral (integer, short, char) and not float or double.

C also provides a set of operators that operate the value of a variable. For example, if we were to increase the value of "var1" by 10, we could do "var1 = var1 + 10" or more crisply, "var1 += 10". Likewise, if we were to multiply the value of "var1" by 10, we could do "var1 = var1 * 10" or "var1 *= 10". Same rule applies to several other operators like subtraction "-"), division ("/"), modulo ("%"), bit-shifts ("<<" or ">>"), logical and ("&"), logical or ("|"), etc.

Another set of handy operators are the unary operators: "++" and "--", these operators increment and decrement an integer variable by 1 respectively. Thus, "var1 = var1 + 1" is same as "var1++". Likewise, "var1 = var1 -1" is same as "var1--".

However, we should note that the above unary operations can be applied either before or after a variable and accordingly, it can have different meanings. For example, "var1++" means that we use the variable first and then increment it, where as, "++var1" means we increment the variable first and then use it. Thus, "var2 = ++var1" and "var2 = var1++" mean different assignments to var2. It is a bad idea to have complicated expressions involving these operators; overzealous usage of these operators can create confusing statements that will reduce code-readability!

Let us use a simple example to illustrate the usage of above operators.

```
#include <stdio.h>

int main () {
    int var_num1, var_num2, var_num3;

    var_num1 = 10;
    var_num1 += 100;
    printf("[Line: %2d] var_num1 is: %d\n", __LINE__, var_num1);

    var_num1 = 10;
    var_num1 *= 10;
    printf("[Line: %2d] var_num1 is: %d\n", __LINE__, var_num1);

    var_num1 = 10;
    var_num1 %= 7;
    printf("[Line: %2d] var_num1 is: %d\n", __LINE__, var_num1);

    var_num1 = 10;
    var_num1 /= 2;
    printf("[Line: %2d] var_num1 is: %d\n", __LINE__, var_num1);

    var_num1 = 10;
    /* Assign before incrementing var_num1 */
    var_num2 = ++var_num1;

    var_num1 = 10;
    /* Assign after incrementing var_num1 */
```

```
    var_num3 = var_num1++;

    printf("[Line: %2d] var_num2 is: %d, var_num3 is : %d\n",
          __LINE__, var_num2, var_num3);
    return 0;
}
```

We provide the output of the above program below. Please note that the (built-in) macro __LINE__ prints the current line of where __LINE__ is present.

```
$ gcc operations.c  -o operations
$
$ ./operations
[Line:  8] var_num1 is: 110
[Line: 12] var_num1 is: 100
[Line: 16] var_num1 is: 3
[Line: 20] var_num1 is: 5
[Line: 31] var_num2 is: 11, var_num3 is : 10
```

C also provides advanced storage types like arrays and strings. In the following sections, we provide a brief introduction for both arrays and strings. We will revisit both of them later.

Arrays

A C array is a series of data, all of them being of the same type. When we define an array, we need to provide a name, the storage type of array elements and the number of the elements. Thus, if we say "int painting_array[1000]", then C defines an array with name painting_array that holds 1000 integer values.

Each array element is identified using an index. For an array with n values, the first element has an index of 0, the second element has the next index value of 1, and in the end, the last element has an index of (n-1). We can easily navigate, and update the integer values stored in this array.

The following figure shows an array (with name painting_array) that has n elements (for sake of simplicity, the keep values of all elements as 0).

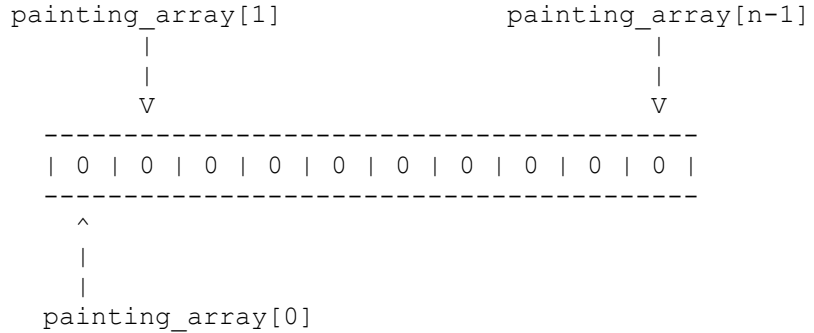


Figure: An Array with n elements

We provide a small example that shows the usage of an array. The program begins by declaring "int painting_array[5]", which means that painting_array is an array of 5 integer elements. This program uses a "for" loop; put simply, a "for" loop helps us traverse all the elements of the array. We will revisit the "for" loop later.

```
#include <stdio.h>

int main () {
    int i;
    int painting_array[4]; /* Define the array */

    /* Assign values to array elements */
    painting_array[0] = 1000;
    painting_array[1] = 1001;
    painting_array[2] = 1002;
    painting_array[3] = 1003;

    for (i=0; i < 4; i++) {
        printf("[i: %d] painting id: %d \n", i, painting_array[i]);
    }

    printf("Length of array: %d \n",
        sizeof(painting_array)/sizeof(painting_array[0]));
    return 0;
}
```

The above example uses the sizeof() function to print the size of the array. First, it uses this function to find the total storage of the array and then, it uses this function again to find the storage of the first element. Next, it divides the storage of the total array with the storage of the first element to get the total number of elements in the array. Note that since all elements are of the same size, it does not matter if we take the size of the first element or any other. Here is the output:

```
$ gcc arrays.c -o array
$
$ ./array
[i: 0] painting id: 1000
[i: 1] painting id: 1001
[i: 2] painting id: 1002
[i: 3] painting id: 1003
Length of array: 4
```

Strings

A C string is an array of char types. As we saw earlier, a char type requires a storage of one byte. So, a string is essentially an array of 1 byte-sized elements. To mark the end of a string, C strings have the last character as '\0', which is a NUL termination. The NUL termination has an ASCII value of 0 and should not be confused with the value of '0' character, which has an ASCII value of 48!

Once again, we provide an program to illustrate C Strings. The example begins by defining a string, var_string; the var_string is defined with "[]" to indicate that it is an array. The program also initializes the var_string with characters ("Mona Lisa"). Next, we find the length of the array and provide this as a limit to the "for" loop. After that, the program uses a "for" loop to iterate through the character elements of the string, var_string.

The example calculates the size of the string, var_string, using two methods. The first method is our earlier method, where we find the storage of the array and divide it by storage of the first element -- this method will include the NUL termination character as well. The second method is by using a built-in function strlen(); this function returns length of a string and we need to include the "string.h" library header to use this function. The strlen() does not include the NUL termination character.

Here is the example:

```
#include <stdio.h>
#include <string.h>

int main () {
    int counter, size;
    char var_string[12] = "Mona Lisa";

    size = sizeof(var_string)/sizeof(var_string[0]);
    for (counter=0; counter < size; counter++) {
        printf("[i: %d] var_string: %c \n", counter, var_string[counter]);
    }

    printf("The length of this string is %d \n", size);
    printf("The length of this string is %d \n", strlen(var_string));
    return 0;
}
```

Note that, in the above definition of var_string, we do not need to provide the number of elements in this array. This is allowed by C when we initialize an array with elements. C uses the type of elements (char in this case) and then allocates enough space to accommodate the passed initial value ("Mona Lisa" in this case). Thus, "char var_string[] = "Mona Lisa";" would have worked equally well!

the output (provided below) shows that each element stores one of the characters of the string, "Mona Lisa". Note that the characters (after "i" equals 8) have the NUL termination and hence print empty characters.

```
$ gcc string.c -o string
$
[i: 0] var_string: M
[i: 1] var_string: o
[i: 2] var_string: n
[i: 3] var_string: a
[i: 4] var_string:
[i: 5] var_string: L
[i: 6] var_string: i
```

```
[i: 7] var_string: s
[i: 8] var_string: a
[i: 9] var_string:
[i: 10] var_string:
[i: 11] var_string:
The length of this string is 12
The length of this string is 9
```

C Basics: Defining Constants

Sometimes, we want a variable to have the same value throughput the program. C allows us to define such variable that maintain their "constantness" throughout the life of the program. We can do this using a handful of ways: (a) using const (short for constant) keyword, (b) using compile-time macros, and (c) using enums (short for enumerations).

It is an unwritten convention to keep names of const-qualified variables, macros, and enum values in upper cases, thereby indicating that they are not regular variables. This convention, of course, is not a language requirement. If you do not follow this convention, then you are bound to irritate your fellow programmers, when they read or review your program!

Constants

One way to define constant values is to use the "const" keyword. All we have to do is to put this keyword before the storage type when defining the variable. Thus, we can define a constant integer variable as: "const int TEN = 10". Once defined, changing the value of this constant would be an error and the compiler would remind you of that!

With that, let us write a simple program that does addition and multiplication using constants. Here we define TEN and FIVE as constants. When run, the output of this program is "Sum is: 15, Product is : 50".

```
#include <stdio.h>

int main () {
    const int TEN = 10;
    const int FIVE = 5;

    int var_sum, var_product;

    var_sum = TEN + FIVE;
    var_product = TEN * FIVE;
    printf("Sum is: %d, Product is : %d\n", var_sum, var_product);
    return 0;
}
```

Macros

Another way to define constants is to have macros using the "# define" semantics. Thus, to define a constant for a painting name, Mona Lisa, we can do this: "#define MONA_LISA 10".

Macros work by doing a simple substitution during compilation stage; to be accurate, macros are parsed just before compilation. This means that if a program contains the macro MONA_LISA, then the parser will simply replace all occurrences of MONA_LISA with the value 10. Since macros are constants, it would also be a compilation error to assign "MONA_LISA = 20" after its value has been assigned initially.

Let us modify our earlier program of addition and multiplication and now make use of macros. The output for this program (provided below) is same as that of the earlier program.

```
#include <stdio.h>

#define  TEN    10
#define  FIVE   5

int main () {
    int var_sum, var_product;

    var_sum = TEN + FIVE;
    var_product = TEN * FIVE;
    printf("Sum is: %d, Product is : %d\n", var_sum, var_product);
    return 0;
}
```

Extra care must be taken when writing non-trivial macros since otherwise a simple replacement may yield unintended consequences. Let us show an example of this erroneous behavior with an incorrect usage of macros. To demonstrate this behavior, the following program replaces TEN with "6 + 4" instead of "10".

```
#include <stdio.h>

#define TEN 4 + 6
#define FOUR 4

int main () {
    int sum, product;

    sum = TEN + FOUR;
    product = TEN * FOUR;
    printf("Sum is: %d, Product is : %d \n", sum, product);
    return 0;
}
```

When we run the above program, the output is: "Sum is: 15, Product is: 26". The sum looks alright but the product is clearly incorrect. What has happened is that the compiler replaced "var_product = TEN * FIVE" with "var_product = 6 + 4 * 5". The expression "6 + 4 * 5" evaluates to "6 + 20" (or 26) because multiplication has higher precedence than addition and hence it is performed before addition.

One way out of this mess is to use parenthesis. Thus, defining TEN as "#define TEN (4+6)" would avoid this problem. In any case, we should restrict using macros to simpler cases.

Enums

Yet another way to define a set of closely related constants is to use an enumeration (or enum for short). If we have a set of variables that are closely related, then we can use one enum to define all of them in one shot. C treats enums values as integers.

Let us use a simple program to understand the behavior of enums. This program (provided below) defines an enum, painters, that holds enumeration for several painters.

```
#include <stdio.h>

#define FIRST_NAME_VALUE 10

enum painters {
    VINCI = FIRST_NAME_VALUE,
    GOGH,
    PICASSO,
    MONET,
};

int main () {
    enum painters value_of_vinci = VINCI;
    int value_of_monet = MONET;

    printf("Vinci: %d, Gogh: %d, Picasso: %d, Monet: %d.\n",
        VINCI, GOGH, PICASSO, MONET);
    return 0;
}
```

The above program defines two variables, value_of_vinci and value_of_monet, and assigns values to them from the enum. Since enums are integers, we can keep the storage size as either "enum painters" or "int". Both would work!

The output of above program is: "Vinci: 10, Gogh: 11, Picasso: 12, Monet: 13.". The enum assigns FIRST_NAME_VALUE to the first element, vinci, and then sequentially keeps assigning the next integer value to the rest.

If we were to define the FIRST_NAME_VALUE as 0 or to not initialize the "vinci" element at all, then the values would start with zero and the output would have been: "Vinci: 0, Gogh: 1, Picasso: 2, Monet: 3.". If we do not need to associate these enums with a hard-coded value, then we should not bother assigning any specific value; the values (starting with 0) would be assigned automatically.

In fact, we can get more creative with enums!

If we were to define "FIRST_NAME_VALUE" as a negative value, let us say -3, then the values assigned would be "-3,-2,-1, 0". Like the case before, enum assigns the value of -3 to the first value and sequentially keeps assigning the next integer value to the rest.

Besides the first element, we can also assign values to other elements and if needed, to more than one elements. Thus, if we were to initialize PICASSO with 1, then the values would be {0,1,1,2}. This is because VINCI/GOGH would receive 0 and 1 and by the time the counter comes to picasso, the value is once again initialized to 1, so it would restart counting as 1 and 2 for PICASSO and MONET.

Clearly, initializing a middle value might make these values non-unique (because now, both gogh and picasso have a value of 1). If we are looking for unique value to these enums (which we probably are!), then we should avoid initializing enums in the middle.

Lastly, we can also use typedef to define a new type of type that can accept values only from the enum list. We rewrite the earlier example and use typedef to define painter as a new type. The output for this program is same as before, so we omit it.

```
#include <stdio.h>

#define FIRST_NAME_VALUE 10

typedef enum {
    VINCI = FIRST_NAME_VALUE,
    GOGH,
    PICASSO,
    MONET,
} painter;

int main () {
    painter value_of_monet = MONET;
    printf("Vinci: %d, Gogh: %d, Picasso: %d, Monet: %d.\n",
        VINCI, GOGH, PICASSO, MONET);
    return 0;
}
```

C Basics: Operator Precedence

When a C expression has multiple operations (it happens more often than not!), then it is possible for some of these operations to be ambiguous. As an example, for expression, "var_num = 6 * 2 + 3", it is not clear if we multiply 6 by 2 first and then add 3 (so that var_num becomes 15) or add 2 to 3 first and then multiply with 6 (so that var_num becomes 30).

To handle such ambiguous cases, C maintains a precedence order for various operators.

We list the precedence order in the table provided below. The list contains some of the common operators and hence, is not exhaustive. We list these operators from top to bottom, in descending precedence -- thus, operators at the top have higher precedence that those that are below it. Also, operators present in the same row have equal precedence.

Operator	Description
(), [], -->, .	Parentheses, Array Index, Pointer member, data structure member
/, %, *,	Division, Modulo, and Multiplication
+, -	Addition and Subtraction
<<, >>	Shift Operators (Left-shift and right-shift)
<, <=, <, >=	Comparison Operators
==, !=	Equality and Inequality
x1 & x2	Bitwise AND
x1 ^ x2	Bitwise XOR
x1 | x2	Bitwise OR
!x	Logical Negation
x1 && x2	Logical AND
x1 || x2	Logical OR
?:	Conditional Ternary Operator
=, +=, -=, /=, *=, %=, &=, ^=, !=, <<=, >>=	Binary Operators

Table: C Operator Precedence

Thus, if an expression has both "*" and "+", then C uses the precedence list to conclude that "*" has a higher precedence than "+". Accordingly, the compiler would execute "*" followed by "+".

Now that we have seen the operator precedence, let us revisit our earlier expression of "var_num = 6 * 2 + 3"! As per the precedence list, "*" has precedence over "+". So, C first multiples 6 and 2 (which yields 12) and then adds 3 to the result. Thus, the value of var_num becomes 15 and not 30.

To help avoid ambiguity, the right thing to do is to use parenthesis. As we can see in the above list, parentheses has a higher precedence than other operators and hence, where ever there is less clarity, we can simply use parentheses to avoid ambiguity. Using parentheses is a good programming practice. As a bonus, it also improves code-readability!

On the other hand, if we wish to do addition first, followed by multiplication, then we can specify that as "var_num = 6 * (2 + 3)". Else, if we intend to multiply 6 and 2 first and then add 3 to the result, then we can specify it as "var_num = (6 * 2) + 3".

Lastly, some of the rows in the table have multiple operators. As we noted earlier, for such rows, the precedence is same for all of them. So, a natural question would be what happens if we have both of these operators together in the same expression. This is okay since the nature of these operators allow us to carry them out in either order. Thus, if we have an expression "10 + 5 - 1", then since both "+" and "-" have the same precedence, we can either add 10 and 5 first followed

by subtracting 1 or we can subtract 1 from 5 first followed by adding 10. The result would be same.

C Basics: printf/scanf

This page provides a brief discussion of some of the common print functions (printf(), sprintf(), and snprintf()) along with the input function (scanf()). These functions are published via the "stdio.h" header file.

printf()

A printf() function takes various arguments, where the first argument determines how many more arguments are present. The first argument consists of format specifiers (like %d, %f) and the printf() replaces all the format specifiers present in the first string with the values that follow. Thus, "printf("This is a trial: %d\n", 100);" would evaluate to "This is a trial: 100" because printf would replace "%d" specifier in the first string with the value of 100.

The printf() function supports formats for various types: (a) %d and %i for integers, (b) %u for unsigned int values, (c) %x and %X for hexadecimal (the upper case prints hexadecimal numbers in upper case), (d) %f for float and double (fractional numbers), (e) %o for octal, (f) %c for char, (g) %s for strings, (h) %p for pointers, and (i) %% for the character % itself.

In fact, printf() function is expressive and allows us to add a lot more details to the basic output type. A generic syntax for printf() function is of the form:
"%[flags][width][.precision][length]type".

The flag can be either "+" or "-". When we specify "+", it means that the printf should print the sign of the numeric value. The "-" sign means something very different -- it represents left padding for the output.

The width attribute specifies the number of decimal values that we want to pad with numbers. For padding (the total characters for the output), we can specify "%10d", which means that the total output character would be 10. By default, padding is done on the left side. We can specify "%-10d" to specify padding on the right side.

The precision attribute specifies the number of decimal values that we wish to print for fractional numbers. Thus, a value of "%.4f" means that we should print 4 decimal values for the float. If the input has additional decimal values, then it is truncated.

The length attribute allows us to tell printf, if we are passing a value that has a more specific length. For example, passing "l" means the value is a long integer and passing "ll" means the value is a long long integer.

With that, let us see an example that prints various types of variables using printf(). It also illustrates both padding and the number of decimals that we want to print for a float.

```
#include <stdio.h>

int main() {
    float var_float = 100.12345;
    int var_int = 1000;

    /* Left-padding of 20 characters and printing 4 decimal values */
    printf("The value of var_float is %20.4f\n", var_float);

    /* Right-padding of 20 characters and printing 4 decimal values */
    printf("The value of var_float is %-20.4f\n", var_float);

    /* Left-padding of 10 characters and printing 1 decimal values */
    printf("The value of var_float is %10.1f\n", var_float);

    /* Left-padding of 20 characters and need to print with the sign */
    printf("The value of var_int is %+20d\n", var_int);

    /* Printing 1 decimal value for an integer; basically the integer itself
*/
    printf("The value of var_int is %.1d\n", var_int);
    return 0;
}
```

Note that for the last part, "%.1d" means printing one decimal value from an integer. However an integer would not have a decimal value, and so "%.1d" is essentially same as "%d". Here is the output:

```
The value of var_float is             100.1235
The value of var_float is 100.1235
The value of var_float is      100.1
The value of var_int is                +1000
The value of var_int is 1000
```

It is also possible to pass the first string to printf in multiple lines. We can use this form if the first string is too long to fit on a single line. For this case, we can split the first string into multiple strings and place them in subsequent lines with no commas between these lines. Here is an example:

```
#include <stdio.h>

int main() {
    float var_float = 100.12345;
    int var_int = 1000;
    char var_str[] = "Mona Lisa";

    printf("The value of var_float is %.4f.\n"
           "The value of var_int is %d.\n"
           "The value of var_str is %s.\n",
           var_float, var_int, var_str);
    return 0;
}
```

And, here is the output:

```
The value of var_float is 100.1235.
The value of var_int is 1000.
The value of var_str is Mona Lisa.
```

sprintf() and snprintf()

Function printf() has two modified variants: sprintf() and snprintf(). These functions allow us to print values to a string instead of the console. Here are their signatures:

```
int sprintf(char *str, const char *format, ...)
int snprintf(char *str, size_t n, const char *format, ...)
```

Both of these functions write the output to the str string; snprintf() is similar to sprintf() except that it writes only n bytes of output the str string.

As usual, let us see an example that demonstrates the usage of these two functions:

```
#include <stdio.h>

int main() {
    int var_int = 1000;
    char str[100];

    sprintf(str, "The value of var_int is %d", var_int);
    printf("[sprintf]: %s\n", str);

    snprintf(str, 10, "The value of var_int is %d", var_int);
    printf("[snprintf]: %s\n", str);
    return 0;
}
```

We provide the output below. We see that since we pass a small size (10 chars) to the snprintf(), the output stores only the first 10 chars of the str variable.

```
[sprintf]: The value of var_int is 1000
[snprintf]: The value
```

scanf()

The scanf() function allows us to receive input from the console/command-prompt. The scanf() function takes address of variables as input and stores the input provided by console at that address.

Let us see an example.

The following code defines two variables, var_int and var_float and uses scanf to read values into these two variables. The scanf function takes "&var_int" and "&var_float" as input; these are pointers and not variables themselves (more on pointers in subsequent pages!). The "&" sign is important; missing it will likely crash your program. Some of us have learned this the hard way!

```
#include <stdio.h>

int main() {
    int var_int;
    float var_float;

    scanf("%d %f", &var_int, &var_float);
    printf("The value of var_int is %d \n", var_int);
    printf("The value of var_float is %-20.4f \n", var_float);
    return 0;
}
```

In the output (which is typically a console), we need to type in inputs to the int and the float; in the output below, the "10000 1000.23232" is the input that we type on the command line. This input should be provided in the same order as expected -- an integer followed by a float. After accepting the input, the program simply prints these values back on the console.

```
$ gcc scanf.c -o scanf
$
$ ./scanf

10000 1000.23232
The value of var_int is 10000
The value of var_float is 1000.2323
$
```

C Conditional Expressions

A program may need to execute different sets of statements depending upon a condition. As an example, if we were to search for a painter of a painting from a list of painters and if it finds the painter, then the program should indicate that it has found the painter. Else, it should continue its search. C handles such cases using conditional expressions.

Let us understand conditional expression using the following flow-chart. The flow-chart starts with a condition that can evaluate to True or False. If the condition evaluates to True, then the program executes task pertaining to that condition otherwise it executes other task. Needless to say, the location of putting a condition expression is dictated by the application logic.

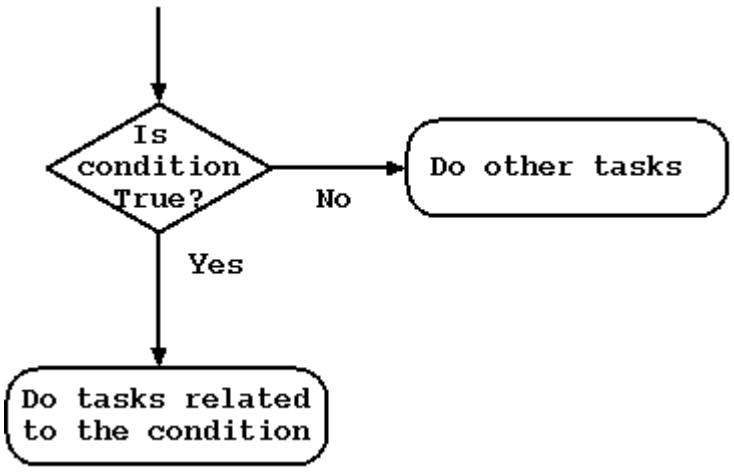


Figure: Conditional Expression in C

C provides four types of conditional expressions: (a) if, (b) if-else, (c) if-"else if"-else, and (d) switch. One can consider the first two forms as simplification of the if-"else if"-else form.

Moving on, we provide generic formats of the first three expressions; we will revisit the generic format of switch expression a little later. For the if-"else if"-else clause, there is no limit to the occurrence of "else if". We show two conditions ("condition1" and "condition2") for "else if" clause, but, a program is free to add as many clauses as it requires.

```
/* Format of a single if expression */
if (condition) {
    Execute some statements
}

/* Format of an if-else expression */
if (condition) {
    Execute statements pertaining to the "condition"
} else {
    Execute Alternative statements
}

/* Format of an if-"else if"-else expression */
if (condition) {
    Execute statements pertaining to the "condition"
else if (condition1) {
    Execute statements pertaining to the "condition1"
else if (condition2) {
    Execute statements pertaining to the "condition2"
} else {
    Execute Alternative statements
}
```

Before we go any further, let us spend some time thinking about what it takes to make the condition evaluate to True or False. We can broadly put such expressions into two categories.

The first category is one that is purely boolean in nature. Let us say, we have a variable, var_x, with a value of 2. Thus, an expression " if (var_x == 2) " would evaluate to True since value of var_x is indeed 2. Further, expressions like " if (x > 0)" or "if (x != 4)" would also evaluate to True. On the other hand, some of the expressions would evaluate to False, like " if (x < 0)" or "if (x != 2)". So, basically, relational expressions that compare a variable for equality, greater than, less than, etc.

The second category is more subtle. C also evaluates expressions with non-zero value as True. Thus, "if (var_x)" would be True because var_x is not equal to zero and so it has some value! As an example, "if (!var_x)" would be False since negation of 2 would be a 0, which is not a value.

A common example for this case is when we have a pointer (we will revisit pointers soon!). Pointers can have a value of NULL and a value of NULL is considered False because NULL means it is pointing to nothing. Thus, if var_ptr has a value of NULL, then the expression "if (var_ptr)" would be False and " if (!var_ptr)" would be True since negation of nothing is, well, something!

Now that we have seen the generic formats and some understanding of what makes a condition True (or False), let us write a simple program and implements these expressions. The program (provided below) handles a simple case where we check the value of a variable, painting_name, against names of various paintings. In the interests of simplicity, we keep painter names as macros and initialize the variable with one of the painting names.

```
#include <stdio.h>

/* Paintings by Leonardo da Vinci and Van Gogh */
#define THE_LAST_SUPPER   1001
#define MONA_LISA         1002
#define POTATO_EATERS     1003
#define CYPRESSES         1004

int main() {
    int painting_name = THE_LAST_SUPPER;

    /* An if clause */
    if (painting_name == MONA_LISA) {
        printf("Mona Lisa was painted by Leonardo da Vinci (1505)\n");
    }

    /* An if-else clause */
    if (painting_name == MONA_LISA) {
        printf("Mona Lisa was painted by Leonardo da Vinci (1505)\n");
    } else {
        printf("Not aware of this painting\n");
    }

    /* An if-"else if"-else clause */
    if (painting_name == MONA_LISA) {
        printf("Mona Lisa was painted by Leonardo da Vinci (1505)\n");
    } else if (painting_name == THE_LAST_SUPPER ) {
        printf("The Last Supper was painted by Leonardo da Vinci (1497)\n");
    } else if (painting_name == POTATO_EATERS) {
```

```
        printf("Potato Eaters was painted by Van Gogh (1885)\n");
    } else if (painting_name == CYPRESSES) {
        printf("Cypreses was painted by Van Gogh (1889)\n");
    } else {
        printf("Not aware of this painting\n");
    }
    return 0;
}
```

Since we initialize the value of painting_name to THE_LAST_SUPPER, the if clause evaluates to False and so, we do not enter the if clause block. For the "if-else" clause, the if case is once again False and so we enter the "else" part and we would see the corresponding output of the printf() call. For the if-"else if"-else clause, all of the expressions evaluate to False, except the second "else if" clause and the output would print the corresponding text.

When we run the program, we find the output provided below.

```
Not aware of this painting
The Last Supper was painted by Leonardo da Vinci (1497)
```

Switch Statements

If the conditions form an if-"else if"-else clause are always integer values and belong to a set of closely-related values, then we can use a special conditional expression, the "switch" statement.

Let us first see a pseudo-code that shows a typical format of the switch expression. A switch handles each clause by comparing to different case values, where each case value must be a constant. If the expression matches any of the case values, then it runs the corresponding the statements; else, it runs the default case (if we have one). We also have a break statement present for each cases. This means that once we are done running the statements, we break out of the switch expression.

```
switch (expression) {
    If needed, declare Variables

    case constant0:
    Execute statements pertaining to the "constant0"
    break;

    case constant1:
    Execute statements pertaining to the "constant1"
    break;

    default:
    Execute Alternative statements
    break;
}
```

It is important to keep two things in mind for a switch statement.

First, the above constant expressions cannot be variables themselves; replacing these case constants, "constant0" or "constant1" with expressions (even if that leads to an integer) would lead to a compilation error. Let us say that we have an integer variable, temp_var that equals 100. In this case, it is not allowed to replace a constant by temp_var, instead they should be 100 or other integer value (often, we use macros or enums to achieve this).

Second, the "expression" should also evaluate to an integer value, otherwise that would also lead to a compilation error. This "expression" value can be either 100 or temp_var since both are of type integer and they both evaluate to an integer value. In addition to an integer, the expression can also evaluate to the related types of short, long, or char.

A note about the default case. It is not mandatory for a switch statement to have a default case but keeping one is a good practice. Basically, the default case can catch all values (perhaps, error cases) that are not covered by individual cases. Alternatively, there could also be scenarios, where most of the cases intentionally go to default and only a few specialized values get the preferential treatment.

Let us see a simple example of a switch statement; this example uses some of the paintings by Leonardo da Vinci and Van Gogh. We define a macro to represent each painting. Typically, if a program uses constants to represent certain states or certain types of data, then they can be represented as macros.

```
#include <stdio.h>

/* Paintings by Leonardo da Vinci and Van Gogh */
#define THE_LAST_SUPPER   1001
#define MONA_LISA       1002
#define POTATO_EATERS           1003
#define CYPRESSES       1004

int main() {
    int painting_name = THE_LAST_SUPPER;

    switch (painting_name) {
        case THE_LAST_SUPPER:
        printf("The Last Supper was painted by Leonardo da Vinci (1497)\n");
        break;

        case MONA_LISA:
        printf("Mona Lisa was painted by Leonardo da Vinci (1505)\n");
        break;

        case POTATO_EATERS:
        printf("Potato Eaters Lisa was painted by Van Gogh (1885)\n");
        break;

        case CYPRESSES:
        printf("Cypreses was painted by Van Gogh (1889)\n");
        break;

        default:
        printf("Not a painting by Leonardo da Vinci/Van Gogh\n");
```

```
        break;
    }
    return 0;
}
```

The output of the above program is "The Last Supper was painted by Leonardo da Vinci (1497)".

We could also make several case statements do the same task. For example, if were to be less specific about the painting, then we could group all the paintings of a specific painter. We can achieve this by removing some of the break keywords and provide the modified program as follows:

```
#include <stdio.h>

/* Paintings by Leonardo da Vinci and Van Gogh */
#define THE_LAST_SUPPER   1001
#define MONA_LISA         1002
#define POTATO_EATERS     1003
#define CYPRESSES         1004

int main() {
    int painting_name = THE_LAST_SUPPER;

    switch (painting_name) {
        case MONA_LISA:
        /* Fall-through */
        case THE_LAST_SUPPER:
        printf("This painting (%d) was painted by Leonardo da Vinci\n",
              painting_name);
        break;

        case CYPRESSES:
        /* Fall-through */

        case POTATO_EATERS:
        printf("This painting (%d) was painted by Van Gogh\n",
              painting_name);
        break;

        default:
        printf("Not a painting by Leonardo da Vinci or Van Gogh\n");
        break;
    }
    return 0;
}
```

The output of the above modified switch program is "This painting (1001) was painted by Leonardo da Vinci".

Please note that it is a good practice to provide the comment "/* Fall-through */" -- this way, new programmers who get to maintain this code would know that the fall-through is intended. God forbid, if they add a break statement in place of fall-through (thinking that it is missing!), then they would break (pun intended!) the program logic.

Conditional Operator

C also provides a handy conditional operator for "if-else" clause: "if (condition) ? do_this_1 : do_this2". If the "condition" is True, then C runs "do_this_1", else, it runs "do_this_2".

A simple example could be to find the maximum of the two numbers; we provide the example below. The output of this program is "The maximum is 10".

```
#include <stdio.h>

int main() {
    int x1 = 5;
    int x2 = 10;

    printf("The maximum is %d\n",  (x1 > x2 ? x1 : x2));
    return 0;
}
```

C Loops: Introduction

Sometimes, an application needs to run in a loop to execute a task multiple times. As an example, if we have an array of paintings and we are looking for a specific painting, then we need to run in a loop and for each run, keep comparing the details of specific painting to that of the loop's current painting; we keep doing this until we find the painting we are looking for.

C provides three different constructs for looping: (1) while loop (2) do-while loop, and (3) for loop. The three loop variants behave differently. So, let us begin by providing a flow-chart that highlights their behavior:

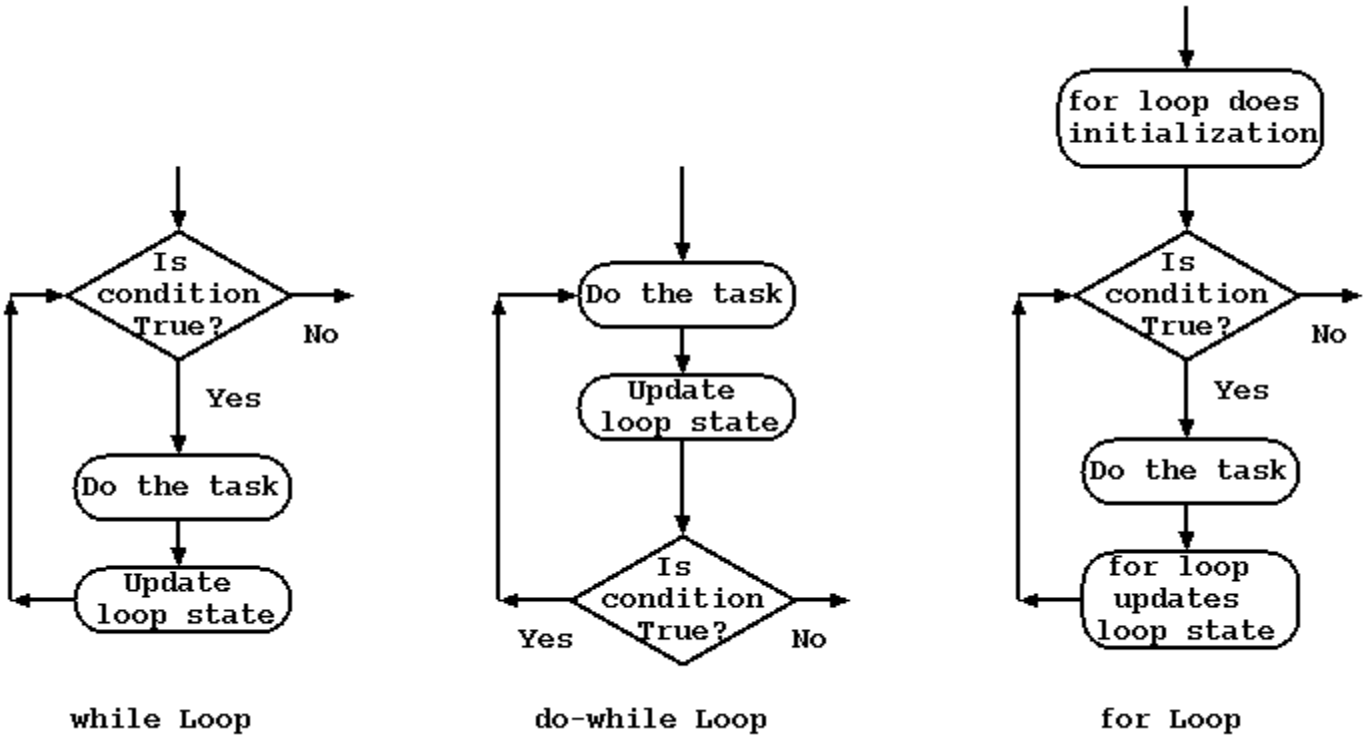


Figure: Three types of C Loops

The while loop is fairly simple. The looping happens as long as a specified condition is true and we quit the loop when the condition becomes false. Naturally, after every round, we need to update the loop state (or the condition variable) so that we can re-check if the condition has become true; in the above example, updating the loop state could be as simple as incrementing a counter variable.

The do-while loop also has a similar format, except that the while clause sits at the very end of the loop block. The distinctive feature of this variant is that the looping block would get executed at least once. This happens because even if the condition is false, the program must run starting from "do" till the point where it reaches the "while" condition. For cases, where we need to run the loop for the first round, whether condition is met or not, we should consider deploying "do-while" method.

The "for" loop cleanly tucks different phases of looping in one expression: initialization, conditional-check, and update conditional variable. The first part of the for loop is the initialization (e.g., "counter=0"), the second part is the loop-termination (e.g., "counter < 3"), and the last part is updating the conditional variable (e.g., "counter++"). Note that both initialization and updating the loop state is done automatically by the for-loop. We just need to specify it in the for-expression.

For while and do-while loops, the above flow-chart shows that we do the task and then update the conditional variable, but the ordering can also be the other way around and should be dictated

by the application logic -- it is possible that in some cases, we update the condition first and then do the task.

Before we proceed any further, let us now take a look at a pseudo-code for these three variants. Please note the semi-colon at the end of the do-while clause!

```
/* A while loop */
while (condition) {
    Do the task
    Update the loop state
}

/* A do-while loop */
do {
    Do the task
    Update the loop state
} while (condition);

/* A for loop */
for (initialize; check condition; update the loop state) {
    Do the task
}
```

Now that we have seen an overview of these three looping variants, it is time to put our learning to test. For that, we can use an example contains a 2-dimensional array such that each row contains a painting ID and its availability in a store. Next, the example prints both of these values for the array using all the three looping styles.

```
#include <stdio.h>

#define THE_LAST_SUPPER 1001
#define CYPRESSES       1002
#define POTATO_EATERS   1003

int main () {
    int counter;
    int painting_array[3][2] = {
            {THE_LAST_SUPPER, 100},
            {CYPRESSES,       10},
            {POTATO_EATERS,   11},
    };

    printf("Let us print using a while loop\n");
    counter = 0;
    while (counter < 3) {
        printf("[i: %d] painting_number: %d Availability: %d \n", counter,
                painting_array[counter][0], painting_array[counter][1]);
        counter++;
    }

    printf("\nLet us print using a do-while loop\n");
    counter = 0;
    do {printf("[i: %d] painting_number: %d Availability: %d\n", counter,
                painting_array[counter][0], painting_array[counter][1]);
```

```
        counter++;
    } while (counter < 3);

    printf("\nLet us print using a for loop\n");
    for (counter = 0; counter < 3; counter++) {
        printf("[i: %d] painting_number: %d Availability: %d\n", counter,
                painting_array[counter][0], painting_array[counter][1]);
    }
    return 0;
}
```

Here is the output. As expected, the output is identical for all the three methods.

```
$ gcc loops.c  -o loops
$
$ ./loops
Let us print using a while loop
[i: 0] painting_number: 1001 Availability: 100
[i: 1] painting_number: 1002 Availability: 10
[i: 2] painting_number: 1003 Availability: 11

Let us print using a do-while loop
[i: 0] painting_number: 1001 Availability: 100
[i: 1] painting_number: 1002 Availability: 10
[i: 2] painting_number: 1003 Availability: 11

Let us print using a for loop
[i: 0] painting_number: 1001 Availability: 100
[i: 1] painting_number: 1002 Availability: 10
[i: 2] painting_number: 1003 Availability: 11
```

Run till Infinity

Sometimes, however, we need to run a program forever in a loop. For example, a network program (say a web-server) may continuously wait for requests from other applications, an interactive program may continuously wait for user-input, or a network protocol may continuously wait for a handshake message from a new peer. Such programs need to keep running and hence, require a loop with a condition that should always evaluates to True! Of course, an infinite loop could also terminate if an error occurs or if we kill the program. But that would be a different story!

All of the above three loop variants can be easily made to run till infinity.

For the while and do-while loops, we can keep the while condition always True; we can do this using "while (1)", because "1" being non-zero is always true. For the for loop, we can skip providing any loop termination clause; we can do this using "for(;;)". With this change, there is no termination-condition and the loop will run happily forever!

C Loops: Continue/Break

C provides methods for an application to be more selective when running inside a loop. There are two specific tasks that an application might wish to do when present in a loop: skip some of the records and break out of the loop. C provides two keywords, "continue" and "break" that allows a program to skip a record or to break out of the loop respectively.

Here is a representative flow-chart that highlights the behavior of these statements. When condition for continue is met, then we skip the rest of the tasks in that iteration of the loop and continue the next iteration of the loop. On the contrary, when the condition for break is met, then we skip the loop altogether and break out of it.

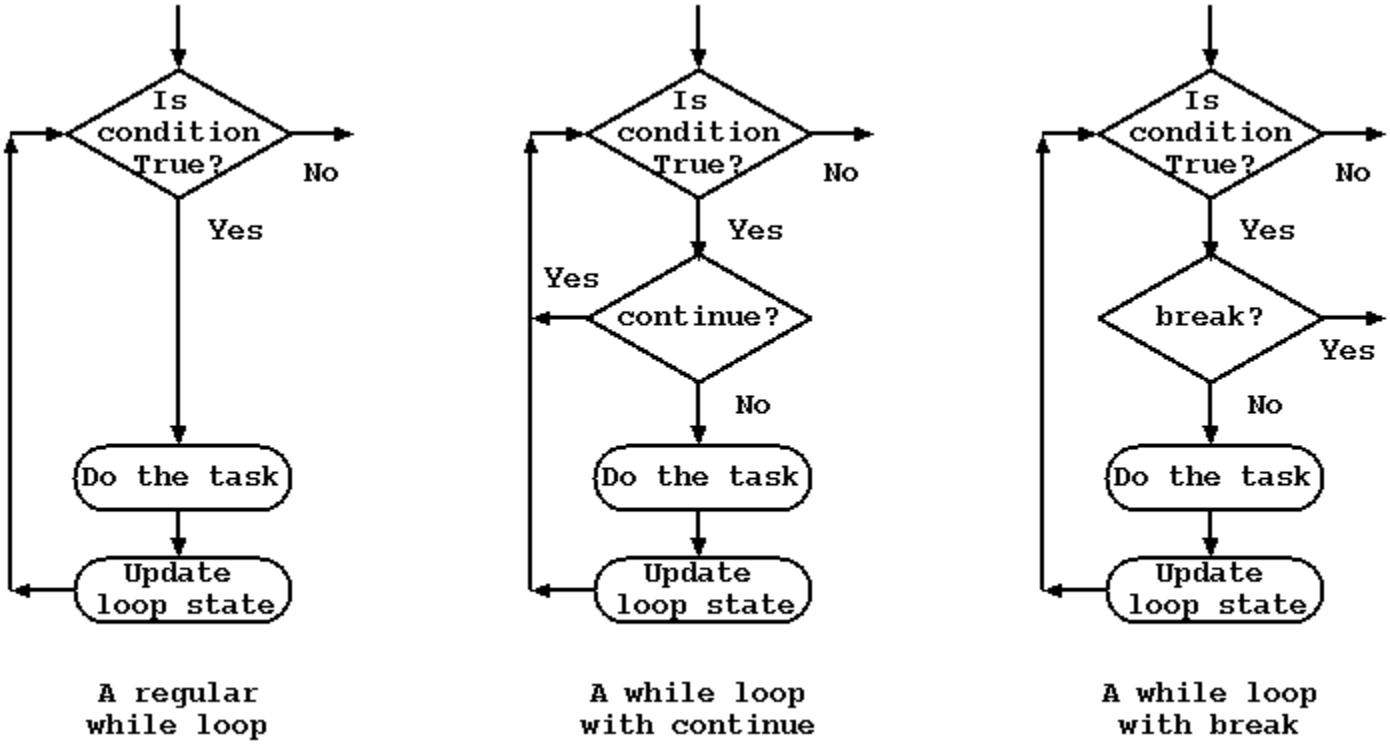


Figure: Continue and Break for a while loop

For cases where we have multiple levels of loops (meaning, a loop running inside another loop), the "break" statement merely exists from the current loop level. The immediate outer loop surrounding the current loop continues to run. Similar behavior holds true for the continue statement as well.

With that, let us go through an example that uses these keywords. In the interests of simplicity, we use only a "for" loop. But, these keywords would apply equally well to the remaining loop variants of while and do-while.

In this example, we need to print an inventory of various paintings and if a particular painting does not have any availability, we skip printing that record. To do so, we take the help of the

"continue" keyword. Another use-case would be when we need to look for a specific painting, let us say, "Mona Lisa" and we want to see its availability. We can run in a loop and once we find "Mona Lisa", we can break out of the loop using the "break" keyword.

```
#include <stdio.h>

#define THE_LAST_SUPPER 1001
#define MONA_LISA       1002
#define POTATO_EATERS   1003
#define CYPRESSES       1004

#define MAX_PAINTINGS         4

int main () {
    int counter;
    int painting_array[MAX_PAINTINGS][2] = {
       {THE_LAST_SUPPER, 100},
       {MONA_LISA,        10},
       {POTATO_EATERS,     0},
       {CYPRESSES,        11},
    };

    printf("Let us print inventory \n");
    for (counter=0; counter < MAX_PAINTINGS; counter++) {
        if (painting_array[counter][1] == 0) {
        /* Means, the painting is not available */
            continue;
        }
        printf("[continue loop] i: %d, painting_id: %d Availability: %d \n",
                counter,
                painting_array[counter][0],
                painting_array[counter][1]);
    }

    printf("Let us search for Mona Lisa \n");
    for (counter=0; counter < MAX_PAINTINGS; counter++) {
        printf("[break loop] i: %d, painting_id: %d Availability: %d \n",
                    counter,
                    painting_array[counter][0],
                    painting_array[counter][1]);
        if (painting_array[counter][0] == MONA_LISA) {
            break;
        }
    }
    return 0;
}
```

The first "for" loop skips paintings that are not available. The second "for" loop searches for "Mona Lisa" and breaks out of the loop when it finds the painting record for "Mona Lisa".

Here is the output:

```
$ gcc continue-break-loops.c -o cbloops
$
```

```
$ ./cbloops
Let us print inventory
[continue loop] i: 0, painting_id: 1001 Availability: 100
[continue loop] i: 1, painting_id: 1002 Availability: 10
[continue loop] i: 3, painting_id: 1004 Availability: 11
Let us search for Mona Lisa
[break loop] i: 0, painting_id: 1001 Availability: 100
[break loop] i: 1, painting_id: 1002 Availability: 10
```

C Functions

Functions are important building blocks of C language (or most of the languages, for that matter). In C, every piece of code sits in one function or the other. In fact, the very first place where the execution begins is in the main() function. All other programs are either called by main() or are called by the functions that are called by the main(), and so on.

Functions offer several key advantages. First, they allow a programmer to keep the program modular -- each function can provide a set of code that typically does one task or a set of closely related tasks. Second, functions promote code-reuse. Different parts of program may need to do the same task and if we have a common function, then all they have to do is to simply call the common function. Third, since the common task now sits in one function, this automatically provides a consistent behavior to all the callers of that function.

We start by providing a pseudo-code for functions and then discuss its components. The pseudo-code (provided below) contains a function named foo(). The function has been intentionally simplified to help us understand how functions work.

```
/* Function declaration */
int foo (int bar);

/* Function definition */
int foo (int bar) {
    int output;

    /* Function Statements */
    ...
    ...

    /* If needed, return a value */
    return (output);
}
```

The pseudo-code starts by declaring the function as "int foo(int bar);". This declaration helps other functions know that there is a function named foo(). The declaration also provides a signature (or the interface) of the function. What follows the declaration is the main body of the function, also known as function definition. A definition holds a set of statements that provide the function logic; these statements are enclosed within a set of curly braces.

A function signature also contains both the list of arguments and the return value of the function. For a function to do anything useful, it often takes a set of input (called arguments or parameters)

from the caller function, as in "int bar". Next, the function statements do some work on that input. And when done, the function returns the output -- in many cases, the output is nothing but a value that the function returns (using the return keyword), as in "return (output)". A function can take as many input values as needed but it can return only one value! We can visualize a function to be a black-box -- we pass some values to it and in return, it provides an output.

With that, it is time to get our hands dirty and write an example to see how functions work. This example (provided below) implements a function (get_painting_year()) that retrieves the year in which a painting was painted. This function gets called from the main() function.

```
#include <stdio.h>

/* Paintings by Leonardo da Vinci and Van Gogh */
#define MONA_LISA            1002
#define POTATO_EATERS        1003

#define YEAR_MONA_LISA       1505
#define YEAR_POTATO_EATERS   1885

#define MAX_ELEM             2

/* Function declaration */
int get_painting_year(int id);

/* Function definition */
int get_painting_year (int param_id) {
    int ret_val;

    printf("\t[%s] Passed param: %d\n", __FUNCTION__, param_id);
    switch (param_id) {
    case MONA_LISA:
        ret_val = YEAR_MONA_LISA;
        break;
    case POTATO_EATERS:
        ret_val = YEAR_POTATO_EATERS;
        break;
    default:
        ret_val = -1;
    }
    printf("\t[%s] All done (returning %d)\n", __FUNCTION__, ret_val);
    return ret_val;
}

int main () {
    int painting_year, i;
    int arr_id[MAX_ELEM] = {MONA_LISA, POTATO_EATERS};

    printf("[%s] Starting to run..\n", __FUNCTION__);
    /* Loop over all the painting names */
    for (i=0 ; i < MAX_ELEM; i++) {
        printf("[%s] i: %d  painting id: %d\n", __FUNCTION__, i, arr_id[i]);
        painting_year = get_painting_year(arr_id[i]);
        printf("[%s] painting year: %d\n", __FUNCTION__, painting_year);
    }
```

```
    printf("[%s] All done..\n", __FUNCTION__);
    return 0;
}
```

The above example starts with function declaration: "int get_painting_year(int id);". This declaration specifies that get_painting_year() takes an integer as an argument and it returns an integer value.

Next, the example provides the function definition. The function uses a switch statement to find the year for a given painting and then returns this value. If the painting does not belong to the list of cases, then the switch statement falls to the default case, where it returns -1.

Lastly, we have the definition of the main() function. Even though the main() is also a function, it never needs a declaration! The reason why we do not need a declaration for main() is that the definition of the main() function is well-specified, as in "int main(int argc, char *argv[])". The combination of argv and argc specify how one can pass command line arguments to the program: argc is the number of strings contained in the argv. One can also omit these arguments entirely, if we do not need to pass any command-line options. That is precisely why, our definition of main() omits these values. More on argc and argv a little later in this section!

The definition of the main() contains an array of names for various paintings. It uses a for loop to go over each element of the array and for each of the element, it calls get_painting_year(). Lastly, it prints the returned value from these functions. Every C program must have one (and only one) main() function -- it is this function that is run first when we execute the program.

Please note that since the function get_painting_year() is defined before the main() function, in this case, it is not necessary to declare it. However, it is a good practice to do so since this way, the programmer need not worry about the ordering of functions int the given file (or a set of files).

When we compile and run the above program, the output (provided below) shows that the main() gets called first. For each element in the loop, we call get_painting_year() and then print the value returned from it. Please note that the macro __FUNCTION__ (similar to macro __LINE__) prints the current function in which it is present.

```
[main] Starting to run..
[main] i: 0  painting id: 1002
       [get_painting_year] Passed param: 1002
       [get_painting_year] All done (returning 1505)
[main] painting year: 1505
[main] i: 1  painting id: 1003
       [get_painting_year] Passed param: 1003
       [get_painting_year] All done (returning 1885)
[main] painting year: 1885
[main] All done..
```

A function may also return a void (which means nothing!) instead of an integer or any other type. Here is a rewrite of the earlier program that moves the task of retrieving and printing the year from the main to the print_painting_year() function. With this, the function does not need to

return any integer. Accordingly, we update both the declaration and the definition of the function!

```
#include <stdio.h>

/* Paintings by Leonardo da Vinci and Van Gogh */
#define MONA_LISA            1002
#define POTATO_EATERS        1003

#define YEAR_MONA_LISA       1505
#define YEAR_POTATO_EATERS   1885

#define MAX_ELEM             2

/* Function declaration */
void get_painting_year(int id);

/* Function definition */
void get_painting_year(int param_id) {
    int ret_val;

    printf("\t[%s] Passed param: %d\n", __FUNCTION__, param_id);
    switch (param_id) {
    case MONA_LISA:
        ret_val = YEAR_MONA_LISA;
        break;
    case POTATO_EATERS:
        ret_val = YEAR_POTATO_EATERS;
        break;
    default:
        ret_val = -1;
    }
    printf("\t[%s] painting year: %d\n", __FUNCTION__, ret_val);
    printf("\t[%s] All done..\n", __FUNCTION__);
}

int main () {
    int painting_year, i;
    int arr_id[MAX_ELEM] = {MONA_LISA, POTATO_EATERS};

    printf("[%s] Starting to run..\n", __FUNCTION__);
    /* Loop over all the painting names */
    for (i=0 ; i < MAX_ELEM; i++) {
        printf("[%s] i: %d  painting id: %d\n", __FUNCTION__, i, arr_id[i]);
        get_painting_year(arr_id[i]);
    }
    printf("[%s] All done..\n", __FUNCTION__);
    return 0;
}
```

The output in this case is same as before except that it is the get_painting_year() that prints the value of the year.

```
[main] Starting to run..
[main] i: 0  painting id: 1002
```

```
       [get_painting_year] Passed param: 1002
       [get_painting_year] painting year: 1505
       [get_painting_year] All done..
[main] i: 1  painting id: 1003
       [get_painting_year] Passed param: 1003
       [get_painting_year] painting year: 1885
       [get_painting_year] All done..
[main] All done..
```

Functions and Scope

Functions provide an important basis for variable scoping in C.

If we define a variable inside a function, then it can be used only inside that function. On the other hand, if we define a variable outside of all functions (including main()), then its scope is global and can be used throughput the file. Thus, depending upon where we define a variable (inside a function or outside), its scope can be different.

As flexible and wonderful as global variables sound, they are not without their share of headaches! When we keep a variable global, we risk it getting overwritten by multiple functions. Also, a global variable is more likely cause conflict when we have to merge our programs (source-files) with additional files when merging different code-bases -- yes, this happens in real-life software processes! So, we should use global variables as sparingly as possible.

As an example, in the earlier program, the function print_painting_year() defines the variable painting_year. This variable can be used only within this function. If we try to use it outside of this function, then that would be a compilation error.

Whenever we call the function print_painting_year(), the value of the variable painting_year is initialized and once the call returns from the function, the variable goes out of scope; that is, it no longer exists in the stack. For some cases, we would like to remember the scope across different invocations.

We can do so by using the "static" keyword that enables the variable to remember its value across different invocations of the function. Thus, defining it as "static int painting_year;" would mean that it would retain the value of painting_year that the function had in the previous invocation.

In fact, functions themselves can also be scope-limited. To do so we can use the same "static" keyword before its definition or declaration -- a static function is a function whose scope is limited to the current source file. Referring to such function beyond the current file would be an error that the compiler would not tolerate!

Thus, if we were to make the print_painting_year() function static, then all we need to do is to add the static keyword at the very start of the function. Here is the function signature; for the sake of brevity, we skip the body of the function.

```
static void print_painting_year (int param_id) {
```

```
    ...
    ...
    ...
}
```

We should reiterate the difference in the behavior of the static keyword, when used with a function and when used with a variable. For a function, it means the scope is only for the current file, where as for a variable, it means that C would remember the value of the variable when we call the function next time.

Recursive Functions

Functions are so versatile that they can call even themselves! Such types of functions are called recursive functions.

A typical use case for such functions is when a function processes a series of data successfully. It starts with the first data element of the series and then calls itself to process the next data in the series, and so on. At each call, the "remaining" series is passed as the argument to the next call of the same function.

To show a use-case, let us consider a function that computes factorial of a number. As we know, a factorial of a number, n is n * (n-1) * (n-2) *..... * 3 * 2 * 1. Thus, factorial of 5 would be 5 * 4 * 3 * 2 * 1 = 120.

Here is a program that does the same. It uses a recursive function, compute_factorial(), to compute factorial of a passed value. Note that since we pass the argument to compute factorial from command-line, the main() function can not omit its arguments (argc and argv) in its definition.

```
#include <stdio.h>

int compute_factorial(int num);

int compute_factorial(int num) {
    int factorial = 1;

    printf("\t[%s] Passed param: %d\n", __FUNCTION__, num);
    if (num != 1) {
        factorial = num * compute_factorial(num-1);
    }
    printf("\t[%s] All done (returning %d)\n",
                    __FUNCTION__, factorial);
    return factorial;
}

int main (int argc, char *argv[]) {
    int number_passed, factorial_value;

    printf("[%s] Starting to run..\n", __FUNCTION__);
    if (argc == 2) {
        number_passed = atoi(argv[1]);
```

```
        printf("Let us calculate factorial for %d\n", number_passed);
    } else {
        printf("Error! Please enter a number.\n");
        return -1;
    }

    factorial_value = compute_factorial(number_passed);
    printf("[%s] The factorial of %s is %d\n",
                    __FUNCTION__, argv[1], factorial_value);
    printf("[%s] All done..\n", __FUNCTION__);
    return 0;
}
```

When we run this program (let us say the executable output is named factorial) for the value 5, it shows how the function calls itself for all the integers starting from 5 to 1.

```
$ gcc factorial.c -o factorial
$
$ ./factorial 5
[main] Starting to run..
Let us calculate factorial for 5
       [compute_factorial] Passed param: 5
       [compute_factorial] Passed param: 4
       [compute_factorial] Passed param: 3
       [compute_factorial] Passed param: 2
       [compute_factorial] Passed param: 1
       [compute_factorial] All done (returning 1)
       [compute_factorial] All done (returning 2)
       [compute_factorial] All done (returning 6)
       [compute_factorial] All done (returning 24)
       [compute_factorial] All done (returning 120)
[main] The factorial of 5 is 120
[main] All done..
```

The output reflects the nature of recursive calls. Recursive programs are run using program stack. Thus, in the first call to compute_factorial(), the program pushes (adds) the context in the stack with the value of num variable as 5. More specifically, the context is the address of the instruction, but we will leave it at that! Since the base case for a factorial is 1, the program calls the same function again, but this time with the value of 4, and thus pushes another context in the stack. It continues to do so with values 3 and 2, till it reaches 1.

And when the call reaches 1, compute_factorial() does not need to call itself since factorial of 1 is 1 and so, it simply returns the same. After 1 completes its task, its context is popped (removed) from the stack. With that, the call-chain in the stack starts to unwind. We go back to the context with value of 2, where it is computed (as 2 * 1) and then it pops as well. In the end, it reaches the call in the stack that was passed with the value of 5. This one also pops and the call is sent back to the main() function.

Inline Functions

C allows us to define inline functions, where the compiler replaces occurrences of functions with the actual code of the function. Thus, inline functions are like macros of functions! We can define an inline function by placing the "inline" keyword before the return type of the function. The "inline" keyword is a request to the compiler and the compiler is free to ignore it.

```
inline void print_painting_year (int param_id) {
     ...
     ...
     ...
}
```

So, why do we need inline functions? Well, if the function is very small, the compiler saves a call (and thus, the time overhead) in the stack by simply substituting the function calls with the function body. Thus, expanding inline functions makes things run faster since there is no need to push or pop function's address and the parameter value on the stack. This optimization can be helpful if we have a small function, that executes frequently.

However, a word of caution with inline functions. Overzealous use of inlining can lead to bloating of code since every function call would simply be replaced by the actual function code. Also, inline functions are not the friendliest functions when we have to debug the code with GDB. Because the compiler substitutes inline functions, the function does not have an address, and so it might be difficult to set GDB breakpoints with the function name. As a workaround, compilers often use additional arguments to make GDB think that even inline functions have an actual address.

Passing arguments to the main() function

The main() function can also accept input from the command line. To do so, we can add arguments to the main() function itself: an integer "argc" representing the number (or count) of arguments and "char *argv[]" representing an array of strings.

In the earlier output for the recursive function, we pass one argument to the program, as in "./factorial 5". The first element of the argv[] array is always the name of the program itself!. Thus, argv[0] would be "./factorial" and argv[1] would be "5". Since the arguments are in string, we need to use atoi() function to convert this string (symbolic as "a" in the atoi() name) to an integer (symbolic as "i" in the atoi() name) and hence the name, atoi!

One can also use the argc/argv arguments to pass explicit options to the program. C allows us to achieve this using the built-in getopt() function.

We provide below an example that uses the getopt() function; we pass "-l" and "-v" to the program and signal the program to do a specific task. The "-l" option prints paintings by Leonardo da Vinci and the "-v" option prints paintings by Van Gogh. Note that we provide multi-line input to printf.

```
#include <stdio.h>

void print_paintings_by_leonardo(void);
```

```
void print_paintings_by_van(void);

void print_paintings_by_leonardo (void) {
    printf("Paintings by Leonardo da Vinci:\n"
           "Mona Lisa and The Last Supper \n");
    return;
}

void print_paintings_by_van (void) {
    printf("Paintings by Van Gogh:\n"
           "Potato Eaters, Cypress, and Starry Nights \n");
    return;
}

int main (int argc, char *argv[]) {
    char c;

    while ((c = getopt(argc, argv, "lv")) != -1) {
        switch (c) {
        case 'l':
            print_paintings_by_leonardo();
            break;
        case 'v':
            print_paintings_by_van();
            break;
        default:
            printf("Incorrect option -- please pass l or v\n");
            break;
        }
    }
    return 0;
}
```

Here is the output of passing these option flags to the program.

```
$ gcc getopt.c -o getopt
$
$ ./getopt -l
Paintings by Leonardo da Vinci:
Mona Lisa and The Last Supper
$
$ ./getopt -v
Paintings by Van Gogh:
Potato Eaters, Cypress, and Starry Nights
$
$ ./getopt -d
./getopt: invalid option -- 'd'
Not a correct option. Please pass options n or p
$
$ ./getopt -lv
Paintings by Leonardo da Vinci:
Mona Lisa and The Last Supper
Paintings by Van Gogh:
$
```

C Arrays

A C array is a series of data, where all of them have the same storage type. Defining an array is simple -- we need to provide the storage type of array element, array name, and its size. Thus, if we say "int painting_array[1000]", then C defines an array, painting_array, that holds 1000 of integer records.

We can access any element using an index. For an array with n values, the first element has an index of 0, the second element has the next index value, 1, and in the end, the last element has an index of (n-1). The indices allow us to easily navigate, and update the integer values stored in this array.

The following figure shows an array, painting_array, with 5 elements.

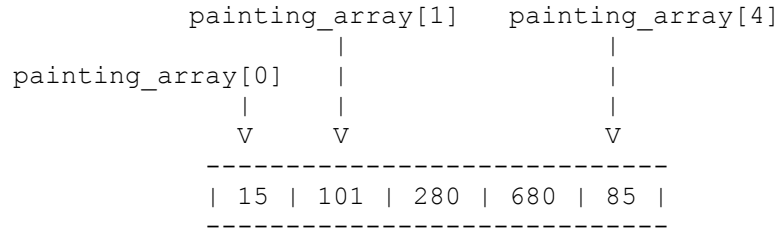


Figure: An Array with 5 elements

Next, we provide a small program to demonstrate how we can define an array and update its elements. The program begins by declaring, painting_array array that holds 4 integer elements.

```
#include <stdio.h>

int main () {
    int painting_array[4]; /* Define the array */
    int i;

    /* Assign values to array elements */
    painting_array[0] = 1000;
    painting_array[1] = 1001;
    painting_array[2] = 1002;
    painting_array[3] = 1003;

    for (i=0; i < 4; i++) {
        printf("[i: %d] painting id: %d \n", i, painting_array[i]);
    }

    printf("Length of array: %d \n",
        sizeof(painting_array)/sizeof(painting_array[0]));
    return 0;
}
```

The above program prints elements of the array along with its size. For printing the size of the array, we use the sizeof() function twice. First, to find the total storage of the array and second,

to find the storage of the first element (since all elements are of the same size, it does not matter if the element is first or any other). After that, we divide the storage of the total array with the storage of the first element and thereby, get the total number of elements in the array.

Here is the output of the above program:

```
$ gcc arrays.c -o array
$
$ ./array
[i: 0] painting id: 1000
[i: 1] painting id: 1001
[i: 2] painting id: 1002
[i: 3] painting id: 1003
Length of array: 4
```

We do not necessarily have to first, declare an array and then, assign values to its elements. C allows us to declare an array and initialize its values, both in the same step. For example, the assignment of values in the previous program can also be achieved as follows: "int painting_array[4] = {1000, 1001, 1002, 1003};".

In fact, specifying the size of the array is optional in the above step. Hence, we could have also written this as: "int painting_array[] = {1000, 1001, 1002, 1003};". With this, C automatically counts the number of elements and assigns a storage size needed to store 4 integers, to the array! What this also means is that, if we were to define an array without specifying neither its size nor initial values, then that would be an error. Thus, compiler would not allow a declaration like "int painting_array[]", since it cannot find the size of the array that it needs to allocate.

Before we move on, we would like to note that when we pass an array to a function, then C passes them by reference, instead of by value. Due to this reason, the sizeof() call would not give the correct size of the array in a called function. The called function only gets a pointer (more on pointers a little later!) and so calling a sizeof() on the passed array would simply return the size of the pointer. If we need the size of the array in the called function, then we must explicitly pass the array length as an additional parameter to the function.

Multi-dimensional Arrays

C also provides multi-dimensional arrays. A multi-dimensional array is essentially an array of arrays. For arrays, there is no limit to the number of dimensions.

Defining a multi-dimensional array is, in some ways, extension of a single-dimensional array. Thus, we can define a 2-dimensional array as "int painting_array[k][m]". This new definition means that the array has k rows and each row has m elements. Thus, the total number of elements is k * m.

Yet another interpretation of "int painting_array[k][m]" is that it is essentially an array of arrays. Thus, painting_array has an array with k values and each of these values is a 1-dimensional array with m values. In fact, this is how C stores a 2-dimensional arrays -- as arrays that in turn have arrays.

Let us see an example of a 2-dimensional array; here, the index to locate an element now has two values: the first value representing the row and the second value representing the column. In other words, "painting_array[i][j] refers to the element sitting on the ith row and the jth column. The following figure shows a 2-dimensional array with 2 rows and 4 columns; we fill each elements with some random numbers.

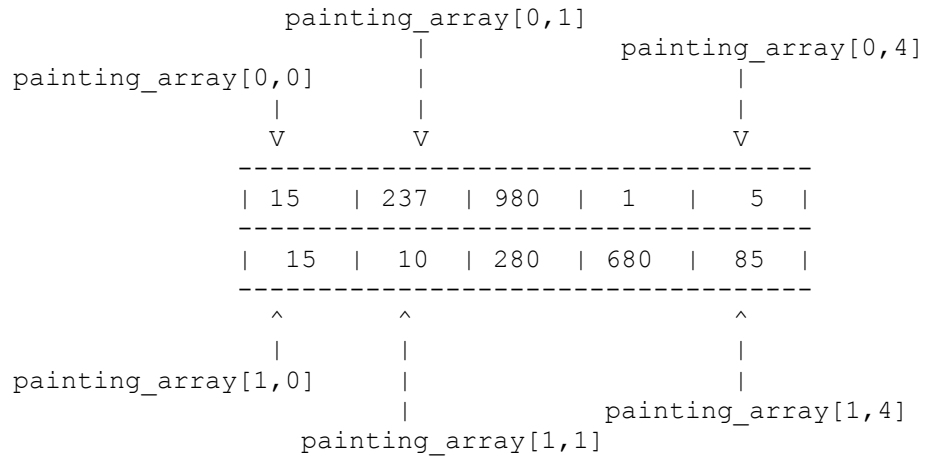


Figure: An 2-dimensional array with 2n elements

With that, let us go ahead and extend our earlier program of one-dimensional array to now use a 2-dimensional array. We modify the program so that the the first row continues to have unique IDs of the paintings and in addition, the (new) second row now has the number of pieces available in the store.

Here is the modified program:

```
#include <stdio.h>

int main () {
    int i;
    int painting_array[4][2]; /* 4 rows, each with 2 columns */

    /* Painting Unique IDs (first column) */
    painting_array[0][0] = 1000;
    painting_array[1][0] = 1001;
    painting_array[2][0] = 1002;
    painting_array[3][0] = 1003;

    /* Number of paintings available in the store (second column) */
    painting_array[0][1] = 100;
    painting_array[1][1] = 10;
    painting_array[2][1] = 10;
    painting_array[3][1] = 11;

    for (i=0; i < 4; i++) {
        printf("[i: %d] painting id: %d Availability: %d\n",
                i, painting_array[i][0], painting_array[i][1]);
    }
```

```
    printf("Length of array: %d\n",
        sizeof(painting_array)/sizeof(painting_array[0][0]));
    return 0;
}
```

When we compile and run the program, we output shows both the painting ID and its availability.

```
$ gcc doublearrays.c -o doublearray
$
$ ./doublearray
[i: 0] painting id: 1000 Availability: 100
[i: 1] painting id: 1001 Availability: 10
[i: 2] painting id: 1002 Availability: 10
[i: 3] painting id: 1003 Availability: 11
Length of array: 8
```

Like the case of one-dimensional arrays, we can also define the above 2-dimensional array and initialize its elements in one shot. For this, we can initialize the array such that we provide each row as a one-dimensional array. This way, the compiler thinks that the array has this many elements, where each element is also an array.

In terms of defining the size of the array, we can do this in two ways. In the first style, we provide both the number of rows and columns in the declaration.

In the second style, we can skip providing the number of rows as long as we provide the number of columns in each row. Providing the column size is key since this helps the compiler know the size of storage for each row. With that, C would automatically count the number of rows from the data provided. Note that failure to provide the number of columns would be a compilation error.

```
/* Style 1: Declare both number of rows and columns */
int painting_array[4][2] = {
  {1000, 100},
  {1001, 10},
  {1002, 10},
  {1003, 11},
};

/* Style 2: Declare only the number of columns */
int painting_array[][2] = {
  {1000, 100},
  {1001, 10},
  {1002, 10},
  {1003, 11},
};
```

Handling Bits: Introduction

Computers store data in binary format. Thus, if we have a decimal number, then we can represent it in a binary format or in a hex format. A char with value of 1 can be represented as

"00000001", where all the bits are zero except the last one. And an integer with value 1 can be represented in binary format as 31 zero bits followed by a single 1 bit. In terms of hex format, an integer with value 1 in a hex format would be "0x00000001" on a 32- bit machine. In this representation, each of the number excluding the initial "0x" represent 4 bits. Thus, we have 8 numbers that represent a total storage of 32 bits.

Manipulate bits allows us a natural way to store information in a compact manner. The constraint here y is that since bit can be either zero or one, the stored data for each bit has to be boolean in nature. Thus, if we have a large number of parameters and if all we care about is whether they are true or false, then using bits to store them would be more efficient.

In terms of decimal value, each of the bit values represent a power of 2. Thus, the last number represents 2 raised to the power of 0 (2^0) or 1, the second to last number represents 2 raised to the power of 1 (2^1) or 1, the third number from the last represents 2 raised to the power of 2 (2^2) or 4, and so on. We assign these values to a bit only if they store 1, else they have no values since they are set to 0. The combined decimal value of all bits is essentially the sum of individual values of these bits. Thus, if we have a binary number with 8 bits as "00000011", then its total sum is sum of values of the bit that store 1. Thus, it would be the sum of the values of the last two bits: (2^1 + 2^0) or (2 + 1) or 3.

Before we go any further, let us look at binary representation and hex representation of a few integer values; we provide these values in the following table. Note that this is for a 32-bit machine, where an integer requires 4 bytes. On a 64-bit system, this representation would require 8 bytes, but the underlying logic would be the same. To elaborate further, let us pick an example. The decimal value 1024 has 11th bit set from the right -- its binary representation reflects the same. For hex representation, we would need to know the decimal value of "0100", which is the 3rd 4-bit set from the right. Since "0100" equals 4, we place the value of 4 in the third 4-bit set from the right.

Decimal	**Binary Representation**	**Hex Representation**
0	00000000 00000000 00000000 00000000	0x00000000
1	00000000 00000000 00000000 00000001	0x00000001
2	00000000 00000000 00000000 00000010	0x00000002
3	00000000 00000000 00000000 00000011	0x00000003
4	00000000 00000000 00000000 00000100	0x00000004
8	00000000 00000000 00000000 00001000	0x00000008
32	00000000 00000000 00000000 00100000	0x00000020
1024	00000000 00000000 00000100 00000000	0x00000400

Table: Binary Representation of Decimal Values

One more thing. The above table provides binary representation such that the most significant byte is the first byte -- such systems are also known as big-endian systems. On some systems (x86-processors, for example), it is the other way around; the most significant byte is actually the

last byte. Such systems are known as little-endian systems. On a little-endian system, an integer with value of 1 would actually be stored as 0x01000000, instead of 0x00000001. This is because "01" is the least significant byte and hence it gets stored first.

The good news is that common bit-handling operations are independent of machine endianness. So, we do not have to worry about it, well, mostly! The only place where it starts to matter is if we are looking into the memory and there, we would see values placed as per the system endianness.

Let us write a simple example that prints decimal values in binary representation. The example (provided below) prints bit values (0 or 1) for various values of an integer. It does so using the print_binary() function. This function uses a char array, binary_string, where each element of the array represents the corresponding bit value in the decimal numbers.

This function checks the value of each bit and if it is set, it adds the char '1' to a char array, else it adds the char '0'. This function checks each bit by comparing the number against a mask (BIT_MASK) that has all of its bits zero except the left-most bit. The function does a continuous left-shift of the number and each iteration, the left-most bit represents one of the bits of the number. Doing an AND (using &) with the mask checks if the left-most bit in the number is set or not. We cover both masks and left-shifting in the next section.

```
 #include <stdio.h>

 #define SIZE_OF_INT 32
 #define TOTAL_LEN (SIZE_OF_INT + 3 + 1) /* 3 for white-spaces and 1 for NUL
*/
 #define BIT_MASK  (1 << 31)

 char binary_string[TOTAL_LEN];

 char *print_binary (int num) {
     int i = 0, bits_since_last = 0;

     while (i <= TOTAL_LEN) {
         binary_string[i++] = '0';
     }
     i = 0;
     while (i < TOTAL_LEN) {
         if (num & BIT_MASK) {
             binary_string[i] = '1';
         }
         bits_since_last++;
         num = num << 1;
         i++;
         if (bits_since_last == 8) {
             bits_since_last = 0;
             binary_string[i] = ' ';
             i++;
         }
     }
     binary_string[TOTAL_LEN-1] = '\0';
     return binary_string;
```

```
}

int main() {
    int var_num;

    var_num = 0;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));

    var_num = 1;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));

    var_num = 16;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));

    var_num = 1024;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));

    var_num = 255;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));
    return 0;
}
```

We would like to briefly mention that the print_binary() prints values into the binary_string starting from left-most bit. We do this because we run the example on a little-endian (x86) system! Once again, for most of the bit-handling tasks -- besides printing bits and looking into bit values in the memory -- we should not bother ourselves with endianness.

When we run the example, we find that for the number 0, we do not have any bits set. No surprises there. For numbers 1, 16, and 1024, we see that only 1 bit is set -- this is because each of these numbers are powers of 2. So, any number which can be expressed as 2^n will essentially have only one bit set in the binary representation. On the other hand, for 255 we see that a total of 8 bits are set. The number 255 belongs to a yet another special class of numbers that can be expressed as (2^n - 1). Such number are 1 less than a power of 2. Some of these numbers are 7, 15, 31, 63, 127, and so on.

```
Number:    0 (Bits: 00000000 00000000 00000000 00000000)
Number:    1 (Bits: 00000000 00000000 00000000 00000001)
Number:   16 (Bits: 00000000 00000000 00000000 00010000)
Number: 1024 (Bits: 00000000 00000000 00000100 00000000)
Number:  255 (Bits: 00000000 00000000 00000000 11111111)
```

In the next few sections, we present a handful of examples that allow us to do some of the common bit-related tasks like shifting bits and setting/clearing bits. In the end, we also present an example that counts the number of bits set in a storage.

Shifting Bits

One of the common bit operations is shifting bits. Depending upon the direction of the shift, it can be a left-shift or a right-shift. We represent left-shift with "<<" and right-shift with ">>". Shifting a number n times on the left is equivalent to multiplying that number by 2 raised to the

power of n (2^n). Shifting a number n times on the right is equivalent to dividing that number by 2^n.

Thus, if we were to left-shift the number 1 by 4, as in "1 << 4", then this would be equivalent to multiplying 1 by 2^4 or 16. On the other hand, if we were to right-shift the number 16 by 4, as in "16 >> 4", then that would be equivalent to dividing 16 by 2^4 or 1.

Besides multiplication and division, yet another use of bit-shifting is in creating bit masks. Bitmasks help us read and write the value of bit stored at any specific location.

Let us now go through a simple example and see how left-shift and right-shift work. The example provided below uses the same print_binary() function provided in the earlier example -- to keep things simple, we put print_binary() implementation in a separate file, print-binary.c. Next, we declare an extern of this function the main file, shift-binary.c. We can pass both the files to gcc and gcc would understand that shift-binary.c file is accessing the print_binary() function defined in the print-binary.c file! If you want to run this example without doing all this, you can always copy the print_binary() function definition in shift-binary.c file here and you should be all set!

```
 [user@codingtree]$ cat print-binary.c
 #define SIZE_OF_INT 32
 #define TOTAL_LEN (SIZE_OF_INT + 3 + 1) /* 3 for white-spaces and 1 for NUL
*/
 #define BIT_MASK  (1 << 31)

 char binary_string[TOTAL_LEN];

 char *print_binary (int num) {
     int i = 0, bits_since_last = 0;

     while (i <= TOTAL_LEN) {
         binary_string[i++] = '0';
     }
     i = 0;
     while (i < TOTAL_LEN) {
         if (num & BIT_MASK) {
             binary_string[i] = '1';
         }
         bits_since_last++;
         num = num << 1;
         i++;
         if (bits_since_last == 8) {
             bits_since_last = 0;
             binary_string[i] = ' ';
             i++;
         }
     }
     binary_string[TOTAL_LEN-1] = '\0';
     return binary_string;
 }
 [user@codingtree]$
 [user@codingtree]$ cat shift-binary.c
```

```
#include <stdio.h>

extern char *print_binary (int num);

int main() {
    int var_num = 1;

    var_num = 1 << 4;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));

    var_num = 1 << 10;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));

    var_num = 15 << 8;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));

    var_num = 1024 >> 5;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));

    var_num = 1024 >> 10;
    printf("Number: %4d (Bits: %s)\n", var_num, print_binary(var_num));
    return 0;
}
[user@codingtree]$
```

To run this example, we simple compile both the files together. The example shows that when we take 1 and left-shift it by 4, we see that the bit in the first place (from right) moves left by 5 bits; the final decimal value becomes 16. On the other hand, when we right-shift 1024 by 5, then we see that the bit in the 10th place moves right by 5 bits; the final decimal value equals dividing 1024 by 2^5 or 1024 divided by 32 or 32.

```
[user@codingtree]$ gcc shift-binary.c print-binary.c -o bit-shift
[user@codingtree]$ ./bit-shift
Number:   16 (Bits: 00000000 00000000 00000000 00010000)
Number: 1024 (Bits: 00000000 00000000 00000100 00000000)
Number: 3840 (Bits: 00000000 00000000 00001111 00000000)
Number:   32 (Bits: 00000000 00000000 00000000 00100000)
Number:    1 (Bits: 00000000 00000000 00000000 00000001)
[user@codingtree]$
```

Setting/Clearing Bits

Yet another common operation for bits is setting, clearing, and checking a given bit. This ability allows us to read, write, and check information on a bit-level. Pretty cool, if you ask me! For these operation, we first need to have a bitmask and then use an appropriate (AND or OR) operation with the bitmask. For setting a bit, we use the OR ("|") operation with the bitmask. For clearing a bit, we use inverse of AND ("~&") with the bitmask. For checking if a bit is set, we use AND ("&") with the bitmask.

Let us understand the idea of setting, clearing, and checking a bit using a storage of one byte. Let us say, we have a number that has a size of one byte and has a value of 8. Its binary representation would be "00010000". Now, the requirement is to set the 3th bit from the right.

We can easily do so, if we have a mask such that the bit at the 3th location from the last is 1 and everything else is 0 or "00000100".

If we take the number and OR it with the mask ("00010000") then we would get "00010000 OR 00000100" or "00010100". Since the mask has all bits 0 except the 3d bit, doing an OR with the number results in two things. First, because all other bits, except the 3rd bit, are set to 0, doing an OR has no effect on all those bits. Thus, if the bits present in those locations are 1, then doing an OR with 0 still leaves them as 1. On the other hand, if they are 0, then well, doing an OR with 0 still leaves them as 0. For the 3dd bit, it is a different story -- doing an OR always gives us 1. This is because, if the 3rd bit in the number is 1, then doing an OR of 1 and 1 would give us 1. On the other hand, if the 3rd bit in the number is 0, then doing an OR of 1 and 0 would still give us 1.

Clearing the bit is taking an inverse of the mask and doing an AND operation with the number. Thus, the inverse of our mask "00000100" is "11111011" and when we do an AND with the number, then we get "00010000 AND 11111011" or "00010000". Since the inverse mask has all bits except the 3rd 1 and the 3rd bit is 0, when we do an AND with the number, once again two things happen. For all other bits, doing an AND with 1 yields the same bit since doing "1 AND 0" leads to 0 as well as "1 AND 1" leads to 1. Thus, the other bit remains unchanged. However, when we do an AND with a zero, then the resulting bit is always 0 since both "0 AND 1" and "0 AND 0" lead to 0. Thus, with this approach, the 3rd bit is set to 0.

Checking if a bit is set is doing an AND operation with the mask. Thus, in our example, if we have to check if the 3rd bit is set or not, then we can do by doing an AND with the mask: "00010000 AND 00000100" or "00000000". The resulting value is zero and hence that means the 3rd bit is not set for the number 8. When we do an AND with the mask, then all the bits except the 3rd one are zero and doing an AND with 0 is always 0. So, all the other bits lead to 0. For the 3rd bit, doing an AND with 1 leads to 1 only if the value of the 3rd bit is set to 1. Else, it leads to 0.

With that, let us see our next example that focuses on setting, clearing, and, checking of bits. The example uses two bitmasks, BIT_10 and BIT_20 that focus on the 10th bit and 20th bit from the last, respectively. We start with number 0 and set both of these bits using these masks one by one. Once set, we clear these bits. The example also illustrates how we can check if a certain bit is set or not.

The example provided below uses the print_binary() function provided in the first example. Like the earlier example, we compile this file with print-binary.c file that contains definition of print_binary() function.

```
#include <stdio.h>

#define  BIT_10      (1 << 9)
#define  BIT_20      (1 << 19)

extern char *print_binary(int num);

void check_bit (int num, int mask, char *desc) {
```

```
    if (num & mask) {
         printf("%s is set\n", desc);
    } else {
         printf("%s is NOT set\n", desc);
    }
}

int main() {
    int var_num = 0;

    printf("BIT_10: %6d (Bits: %s)\n",   BIT_10, print_binary(BIT_10));
    printf("BIT_20: %6d (Bits: %s)\n\n", BIT_20, print_binary(BIT_20));

    var_num = var_num | BIT_10;
    printf("Number: %6d (Bits: %s)\n", var_num, print_binary(var_num));
    check_bit(var_num, BIT_10, "BIT_10");

    var_num = var_num | BIT_20;
    printf("Number: %6d (Bits: %s)\n", var_num, print_binary(var_num));
    check_bit(var_num, BIT_20, "BIT_20");

    var_num = var_num & ~BIT_20;
    printf("Number: %6d (Bits: %s)\n", var_num, print_binary(var_num));
    check_bit(var_num, BIT_20, "BIT_20");

    var_num = var_num & ~BIT_10;
    printf("Number: %6d (Bits: %s)\n", var_num, print_binary(var_num));
    check_bit(var_num, BIT_10, "BIT_10");
    return 0;
}
```

Here is the output:

```
 [user@codingtree]$ gcc example3-set-clear-bits.c print-binary.c -o set-
clear-check-bits
 [user@codingtree]$ ./set-clear-check-bits
 BIT_10:    512 (Bits: 00000000 00000000 00000010 00000000)
 BIT_20: 524288 (Bits: 00000000 00001000 00000000 00000000)

 Number:    512 (Bits: 00000000 00000000 00000010 00000000)
 BIT_10 is set
 Number: 524800 (Bits: 00000000 00001000 00000010 00000000)
 BIT_20 is set
 Number:    512 (Bits: 00000000 00000000 00000010 00000000)
 BIT_20 is NOT set
 Number:      0 (Bits: 00000000 00000000 00000000 00000000)
 BIT_10 is NOT set
 [user@codingtree]$
```

Setting bits, clearing bits, and checking bits are popularly used for dynamic systems consisting of various states. For such systems, we can use flags to represent these states as boolean attributes. If we were to choose events in the life of a painting, then we can model it using states or flags. As one example, let us say we want to store if a painting in an art gallery is sold or not. We can

do so using a single flag -- if the flag is sold, then we set a particular bit. Thus, by checking that particular bit, we can know if the painting is sold or not.

In fact, let us extend this example to understand usage of flags a little bit more. If we were to assume, then we can say, let us say 8 life-events in the lifespan of a painting. We use a single flag to reflect each of these events. We have purposely chosen 8 events since we can represent them using 8 bits or one byte! Moving further, we provide below eight flags that correspond to each of the 8 states. In terms of bits, all of these flags have only a single bit set and the bit keeps shifting from right to left.

```
#define PAINTING_WORK_IN_PROGRESS       0x01    /* Bit values: 00000001 */
#define PAINTING_OIL_PAINT_DRYING       0x02    /* Bit values: 00000010 */
#define PAINTING_DONE_BEING_LOADED      0x03    /* Bit values: 00000100 */
#define PAINTING_IN_TRANSIT             0x04    /* Bit values: 00001000 */
#define PAINTING_IN_STORE_WAREHOUSE     0x10    /* Bit values: 00010000 */
#define PAINTING_IN_STORE_DISPLAY       0x20    /* Bit values: 00100000 */
#define PAINTING_IN_GALLERY_DISPLAY     0x40    /* Bit values: 01000000 */
#define PAINTING_SOLD                   0x80    /* Bit values: 10000000 */
```

Now, I know that the above states are a little artificial (you never know!). But, the example shows that we can use a single byte to store as many as 8 states -- thus, each bit of a byte ends up representing one state. If the first bit (from left) is set, then we know that the painting is in a sold state. Same holds for other states.

Since our model is dynamic, the painting switches from one state to another. When a painting moves to a new state, we need to do two things: clear the flag corresponding to the old state and set the flag corresponding to the new state. Here is a code snippet of how we would do that when a painting gets sold from a gallery display.

```
painting_state = painting_state & ~PAINTING_IN_GALLERY_DISPLAY;
painting_state = painting_state | PAINTING_SOLD;
```

Counting Number of Bits Set

Counting the number of bits set in a storage (let us say an integer) is another common task. Hence, we provide a small example that achieves the task using two different approaches.

Both examples use the same print_binary() function provided in the very first example on this page, to print bits of an integer. In the interests of space, we put print_binary() implementation in a separate file, print-binary.c. Next, we declare this function as an extern in the main file, counting-bits.c. We can pass both the files to gcc and gcc would understand that counting-binary.c file is accessing the print_binary() function defined in the print-binary.c file!

The first example takes the obvious route. It keeps shifting the passed number towards the right by one. And with each shift, it does an AND with 1. Thus, if the last bit is 1, then doing an AND with 1 would lead to 1. In that case, it increases the counter. When it is done shifting the number by SIZE_OF_INT times, it returns the counter. Thus, if we have a number, let us say

"00001111", then when as we shift this number, the bit at each position becomes the bit at the first position and doing an AND with 1 (which is "00000001") checks if the last bit is set or not!

```
#include <stdio.h>

#define SIZE_OF_INT   32

extern char *print_binary(int num);

int number_of_bits (int num) {
    int i, total_count = 0;
    for (i = 0; i < SIZE_OF_INT; i++) {
        if (num & 1) {
            total_count++;
        }
        num = num >> 1;
    }
    return total_count;
}

int main() {
    int var_num;

    var_num = 7;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));

    var_num = 8;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));

    var_num = 255;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));

    var_num = 256;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));

    var_num = (1 << 20) - 1;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));
    return 0;
}
[user@codingtree]$ gcc counting-bits.c print-binary.c -o counting-bits
[user@codingtree]$ ./counting-bits
Number:       7 (Bits: 00000000 00000000 00000000 00000111), Bits set:  3
Number:       8 (Bits: 00000000 00000000 00000000 00001000), Bits set:  1
Number:     255 (Bits: 00000000 00000000 00000000 11111111), Bits set:  8
Number:     256 (Bits: 00000000 00000000 00000001 00000000), Bits set:  1
Number: 1048575 (Bits: 00000000 00001111 11111111 11111111), Bits set: 20
[user@codingtree]$
```

The above approach is okay as long as we have to count the number of bits once in a blue moon. But, if we have to do the counting for several million numbers, then this approach will easily

become rather expensive. The reason is that counting all the bits for a number with n bits is O(n). One practical way to reduce the time-complexity is to use a lookup table. Our next example does exactly the same.

For that, we build an array that holds values for a set of 4 bits. Thus, given a number between 0 and 15 (4 bits can have a maximum value of 15), we store the number of bits present in each number in this array: the number 1 has 1 bits, number 2 has 1 bits, number 3 has 2 bits, and so on. With that, the bit-counting becomes a lookup operation as long as we check each 4-bits, one at a time.

For each storage, we read the value of 4 bits and then do a lookup to count the number of bits. The example uses a 4 bit mask with all bits set to 1 to extract the value of each 4-bit. Since we are counting the number of bits set in 4 bits in one step, this approach would be roughly 4 times faster than the earlier one. If we wanted to optimize this further, then we can count the number of bits set in 8 bits in one shot. Of course, for a lookup with 8 bits, we would need a bigger lookup table.

Here is the modified example. Since the output is same as above, we do not provide it.

```
#include <stdio.h>

#define SIZE_OF_INT   32
#define SIZE_OF_BLOCK 4
#define MASK_4BITS  ((1 << SIZE_OF_BLOCK) - 1)

extern char *print_binary (int num);

short arr_bitcount[] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};

int number_of_bits (int num) {
    int i, total_count = 0;
    int total_iterations = SIZE_OF_INT/SIZE_OF_BLOCK;
    unsigned char x;

    for (i = 0; i < total_iterations; i++) {
        x = num & MASK_4BITS;
        total_count += arr_bitcount[x];
        num = num >> SIZE_OF_BLOCK;
    }
    return total_count;
}

int main() {
    int var_num;

    var_num = 7;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));

    var_num = 8;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));
```

```
    var_num = 255;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));

    var_num = 256;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));

    var_num = (1 << 20) - 1;
    printf("Number: %7d (Bits: %s), Bits set: %2d\n",
            var_num, print_binary(var_num), number_of_bits(var_num));
    return 0;
}
```

Handling Bits: Bitmaps

When we use one of the built-in data types, like char or int, for processing bits, we run into the issue of having only a limited number of bits to play with. For example, an unsigned char can give us only 8 bits and an unsigned int can give us only 32 bits (on a 32-bit machine). This is okay as long as we need only a limited number of bits. Sometimes, that is not enough. Often applications might need to handle hundreds of bits and using an int or even a double would no longer suffice. We are going to need a bigger boat!

The workaround for such cases is to use a bitmap. Also known as a bit array, a bitmap allows us to have an array, where each element can hold a built-in storage like an integer or a char and each of these array elements store bit-information. Thus, if we have a bitmap array with 100 char elements, then we have a total of 800 bits since each char element can in turn store 8 bits. A bitmap allows us to manipulate (set or clear) any bit out of these 800 bits as if it were one single variable.

Bitmaps can support all bit-related operations like setting a bit, clearing a bit, and checking if a bit is set, and many more. Since a bitmap stores individual bits in its array elements, each of these operations need to find two offsets. First, the array index of the element that houses the bit. Second, the position of the bit within that array element.

In the above example of 800 bits bitmap, if we wanted to set the 100th bit, then we would need to locate the array offset first. To locate array offset, we can divide the specified bit with the size of the array element, in this case, 8. Thus, the 100th bit would be present in the 100/8 or in the 12th element. Next, to locate the offset within that element, we can use the modulo of the specified bit with the size of the array element. In this case, that would be 100 % 8, which is 4. So, the 100th bit would be the 4th bit sitting in the 12th element.

Next, we provide an example to help us understand bitmaps a little better. The example (provided below) uses an array (bitStream) of unsigned chars and has a total of 6 chars. Thus, this storage equals 48 bits. The implementation offers 4 bitmap operations: setBitmapArray() to set the kth bit (from right-end), clearBitmapArray() to clear the kth bit (from right-end),

checkBitmapArray() to check if the kth bit (from right-end) is set or not, and printBitmapArray() to print all bits of the bitmap. Here is the example:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define NUM_OF_BYTES 6
#define TOTAL_LEN (NUM_OF_BYTES * 8 + 3 + 1)

#define BYTE_COUNT(x) ((x) / 8)
#define BIT_COUNT(x)  ((x) % 8)

char binary_string[TOTAL_LEN];
unsigned char *bitStream = NULL;

char *printBitmapArray () {
    int i, j, temp;
    int index = TOTAL_LEN - 2;

    while (i <= TOTAL_LEN) {
        binary_string[i++] = '0';
    }
    binary_string[TOTAL_LEN-1] = '\0';

    for (i = 0; i < NUM_OF_BYTES; i++) {
        for (j = 0; j < 8; j++) {
            if ((1 << j) & bitStream[i])
                binary_string[index] = '1';

            index--;
        }
        binary_string[index--] = ' ';
    }
    return binary_string;
}

void setBitmapArray (int loc) {
    if (loc < 1) return;
    bitStream[BYTE_COUNT(loc-1)] |= (1 << BIT_COUNT(loc-1));
}

bool checkBitmapArray (int loc) {
    if (loc < 1) return;
    if (bitStream[BYTE_COUNT(loc-1)] & (1 << BIT_COUNT(loc-1)))
        return true;
    else
        return false;
}

void clearBitmapArray (int loc) {
    if (loc < 1) return;
    bitStream[BYTE_COUNT(loc-1)] &= ~(1 << BIT_COUNT(loc-1));
}
```

```
void initBitmapArray(int num_of_bytes) {
    if (!bitStream) {
        bitStream = (unsigned char *)malloc(sizeof(char *) * NUM_OF_BYTES);
        if (!bitStream) return;
    }
    bzero(bitStream, NUM_OF_BYTES);
}

int main() {
    initBitmapArray(NUM_OF_BYTES);

    setBitmapArray(4);
    printf("[Set    4th bit] %s\n", printBitmapArray());
    printf("[Check  4th bit] %d\n", checkBitmapArray(4));

    setBitmapArray(12);
    printf("[Set   12th bit] %s\n", printBitmapArray());
    printf("[Check 12th bit] %d\n", checkBitmapArray(12));

    clearBitmapArray(12);
    printf("[Clear 12th bit] %s\n", printBitmapArray());
    printf("[Check 12th bit] %d\n", checkBitmapArray(12));

    clearBitmapArray(4);
    printf("[Clear  4th bit] %s\n", printBitmapArray());
    printf("[Check  4th bit] %d\n", checkBitmapArray(4));
    return 0;
}
```

The output of the example shows that now we have 6 bytes of storage available and the set, clear, and verify operations work as if this were a regular storage like an integer or a char.

```
[Set    4th bit] 000000 00000000 00000000 00000000 00000000 00001000
[Check  4th bit] 1
[Set   12th bit] 000000 00000000 00000000 00000000 00001000 00001000
[Check 12th bit] 1
[Clear 12th bit] 000000 00000000 00000000 00000000 00000000 00001000
[Check 12th bit] 0
[Clear  4th bit] 000000 00000000 00000000 00000000 00000000 00000000
[Check  4th bit] 0
```

Please note that the example uses an unsigned char as individual storage types, but one can also use an unsigned integer to do the same. However, this may not not buy as a whole lot since we can also add more number of chars in the above example. Thus, if we want a bitmap of 800 bits, then we can either use 100 unsigned chars or use 25 unsigned ints.

Pointers in C: Introduction

C uses pointers to refer to variables by pointing to their address in memory. If we have a variable x of type int, then we can use a pointer to refer to the address of x. When using pointers, we need to remember three basic syntax rules. First, we can define pointers using "*" keyword. Second,

we can refer to the address of a variable by using "&" keyword. Lastly, we can use the "*" operator to retrieve value stored at a given address.

Let us start with a simple example. Here, ptr_to_x is a pointer that points to the address of x. The value of ptr_to_x is, in fact, the memory address of x.

```
int x = 100;
int *ptr_to_x; /* this is a pointer */

ptr_to_x = &x; /* ptr_to_x points to the memory address of x */
```

Since x is of the (storage) type int, ptr_to_x starts at memory location, &x, and then it automatically reads sizeof(int) bytes, starting at this location for finding the value stored at that location.

The following figure shows an example of address and value stored for both of these variables, x and ptr_to_x. As we can see from this figure, the value stored at the pointer (ptr_to_x) is the address of x.

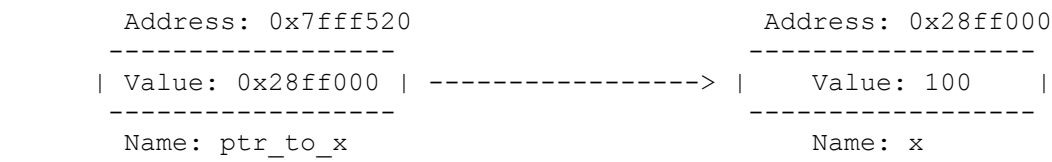


Figure: A pointer stores an address

The above figure also shows that the pointer (ptr_to_x) has a storage of its own. The storage size of the pointer depends upon the system. In practice, on 32 bit processors (e.g. Celeron, Pentium I, Pentinum II, Pentium III etc), a pointer would need 4 bytes of storage and on 64 bit systems (e.g. Intel Core i3, Core i5, Core i7, etc), a pointer would need 8 bytes of storage. We can use the sizeof() function to find out the storage size for a pointer. Thus, in the above example, sizeof(ptr_to_x) would tell the storage size of pointer ptr_to_x.

Let us now see another example, where we use the "*" keyword to access value stored at an address. Then, the example uses the "*" keyword to swap values between two variables. The example also shows that we can have multiple pointers point to the same value.

```
#include <stdio.h>

int main () {
    int x, y, temp;
    int *ptr_to_x, *ptr_to_x1, *ptr_to_x2;

    y = 200;
    x = 100;
    ptr_to_x = &x;
    printf("Value of x: %d and *ptr_to_x: %d\n", x, *ptr_to_x);
    printf("Address of x: %p and ptr_to_x: %p\n", &x, ptr_to_x);

    /* Swapping value of x and y */
```

```
    printf("Before swapping, x: %d  y: %d\n", x, y);
    temp = y;
    y = *ptr_to_x;
    *ptr_to_x = temp;
    printf("After  swapping, x: %d  y: %d\n", x, y);

    /* Making multiple pointers point to the same value */
    ptr_to_x1 = &x;
    ptr_to_x2 = &x;
    return 0;
}
```

Note that we pass the "%p" formatting option to printf() function to print the address pointed by a pointer. Here is the output of the above program:

```
Value of x: 100 and *ptr_to_x: 100
Address of x: 0xbfd44178 and ptr_to_x: 0xbfd44178
Before swapping, x: 100  y: 200
After  swapping, x: 200  y: 100
```

Since a pointer points to the memory location of a variable and since pointers themselves are variables as well, it is also possible to point to the memory location of a pointer itself. We can use the keyword "**" to achieve that. In the following code, ptr_to_ptr_to_x is a pointer that itself point to another pointer, ptr_to_x! Accordingly, we can retrieve the value of x from ptr_to_ptr_to_x using "**ptr_to_ptr_to_x".

```
int x;
int *ptr_to_x;
int **ptr_to_ptr_to_x;

/* ptr_to_x points to the memory address of x */
ptr_to_x = &x;

/* ptr_to_ptr_to_x points to the memory address of ptr_to_x */
ptr_to_ptr_to_ x = &ptr_to_x;
```

When a pointer does not point to any address, its value is a symbolic NULL, which represents an integer value of 0. The compiler would complain bitterly, if we assign a non-zero integer value to a pointers; a value of 0 (or NULL) is the only exception. This special value is used very often for conditional expressions. Thus, if we know that a pointer does not point to anything, then the following expressions would evaluate to True: "if (ptr_to_x == NULL)" or "if (!ptr_to_x)". On the other hand, if it does, then the opposite expressions would evaluate to True: "if (ptr_to_x != NULL)" or "if (ptr_to_x)".

Constant Pointers

C allows us to make the value pointed by a pointer constant. Once the constant value is declared constant, we cannot change it. For this purpose, we can use the "const" keyword. Further, we can use the same keyword to restrict the pointer's address as well. In other words, the "const" keyword allows us to constraint not only the value pointed by the pointer, but also its address.

In fact, there are three possible cases of this "constantness": (a) the value pointed to by the pointer is constant, (b) the address of the pointer is constant, and (c) both value and address of the pointer are constant.

In the first case, we constrain the value pointed by the pointer. The statement, "const int *x" means that the value pointed by the pointer is a constant and hence whatever is stored at that value cannot be changed. Trying to assign the value of the pointer to a new value would surely throw the compiler into a rage. However, the compiler would happily comply, if we reassign the pointer to a new address!

```
int main () {
    int varA = 10, varB = 20;
    const int* x = &varA;

    *x = 20;        /* Compiler would throw an error */
    x = &varB;
    return 0;
}
```

In the second case, we constrain the address of the pointer and with that, we cannot change the address of the pointer. However, changing the value of the pointer would be okay. Thus, in the following code, "x = &varB" would lead to a compiler error, but "*x = 20" would not.

```
int main () {
    int varA = 10, varB = 20;
    int *const x = &varA;

    *x = 20;
    x = &varB; /* Compiler would throw an error */
    return 0;
}
```

In the third case, we crank it up a notch and constraint both the pointer and its value! In the following code, the compiler would not allow the last two assignment statements.

```
int main () {
    int varA = 10, varB = 20;
    const int *const x = &varA;

    *x = 20;   /* Compiler would throw an error */
    x = &varB; /* Compiler would throw an error */
    return 0;
}
```

If it gets confusing as to what location of "const" results in what constraint, then there is a simple trick! If the "const" qualifier is near the "*", then it constrains the pointer (e.g. in "int *const x"), else it constrains the value (e.g. in "const int *x").

Besides qualifying variable, we can also use the "const" keyword when passing pointers (or for that matter values) to a function. Using the "const" keyword, we can specify that the called

function should not modify the value of the passed pointer or the address pointed by the pointer, or both.

So, why do we need to provide such constant pointers or values to function calls? Well, it is possible that the called function might be making a copy of the value of the passed pointer to another pointer and may not need to modify the value pointed to by the passed pointer. Using a const qualifier explicitly, the caller can explicitly tell the called function that, by design, it is not supposed to modify the passed value. C allows us to restrict the called function from modifying the original value by using a "const" qualifier. Here is the signature of one such function that constraints the value of x_copy_from from being modified.

```
int copy_the_value(int *x_copy_to, const int *x_copy_from);
```

It is a good idea to use the "const" keyword, whenever it is applicable. When writing new functions, if the design dictates that the called function should not modify the passed pointer (or its value), then we should not hesitate for a second in making the parameter constant. If you do not believe me, then simply look for "man string.h" in Google (or on your terminal) and you would see that a lot of string and memory functions already use this keyword.

Pointers in C: Pointers and Arrays

Pointers can be easily mapped to an array and can be used to navigate the array elements. Once mapped, navigating to an element of the array is as simple as incrementing the pointer! The mapping works because an array uses a contiguous piece of memory. Thus, if we have a pointer pointing to the first element of the array, then we can increment the pointer to refer to the next element, and so on.

For example, the following code uses a pointer, ptr_to_array_x to point to the first element of an array, array_x.

```
int array_x[100];
int *ptr_to_array_x;

/* ptr_to_array_x points to the first element of the array */
ptr_to_array_x = &array_x[0];

/* Or, even better! */
ptr_to_array_x = array_x;
```

Since ptr_to_array_x points to the first element, *ptr_to_array_x is same as array_x[0]. If we increment the pointer, then it points to the second element; thus, *(++ptr_to_array_x) would be same as array_x[1], and so on.

Let us now see an example (provided below) that shows two ways to print elements of this array -- one by advancing the array index and the other by incrementing a pointer. We can point a pointer to the next element of the array by incrementing it as: "ptr_to_array_x++". Since

ptr_to_array_x is of type int, increasing the pointer automatically moves the pointer to the location of the next int of the array by advancing it by size of an integer.

```
#include <stdio.h>

#define TOTAL_ELEMENTS 5

int main () {
    int arr_x[TOTAL_ELEMENTS] = {1, 2, 3, 4, 5};
    int *ptr_to_arr_x;
    int i;

    /* Traverse the array using its index */
    printf("Let us print using array index\n");
    for (i = 0; i < TOTAL_ELEMENTS; i++) {
        printf("Value: %d Address: %p\n", arr_x[i], &arr_x[i]);
    }

    /* Traverse the array using a pointer */
    printf("\nLet us print using a pointer\n");
    ptr_to_arr_x = arr_x;
    for (i = 0; i < TOTAL_ELEMENTS; i++) {
        printf("Value: %d Address: %p\n", *ptr_to_arr_x, ptr_to_arr_x);
        ptr_to_arr_x++;
    }
    return 0;
}
```

Note that "ptr = ++ptr_to_array_x" is not the same as "y = ++(*ptr_to_array_x)" -- the former means incrementing the address referred to by the pointer and assigning it to a new pointer, ptr. The the latter means increasing the value pointed to by the pointer, ptr_to_array_x and assigning to variable y. You might also run into yet another variation: "y = *++ptr_to_array_x" or "y = *(++ptr_to_array_x)" -- this means that we first increment the pointer and then read the value pointed by it. It is a good idea to use parenthesis,s o that the code becomes more readable.

The output shows that the value and address of each element is identical for both cases. As expected, the output shows that any two consecutive addresses of the array differ by a value of 4, which is the size of an integer; we use a 32-bit machine to compile the code.

```
Let us print using array index
Value: 1 Address: 0xbf936e24
Value: 2 Address: 0xbf936e28
Value: 3 Address: 0xbf936e2c
Value: 4 Address: 0xbf936e30
Value: 5 Address: 0xbf936e34

Let us print using a pointer
Value: 1 Address: 0xbf936e24
Value: 2 Address: 0xbf936e28
Value: 3 Address: 0xbf936e2c
Value: 4 Address: 0xbf936e30
Value: 5 Address: 0xbf936e34
```

In the end, a word of caution about using sizeof() function when we refer to arrays as pointers! We should not the sizeof operator on painting_array pointers since it would not give us the size of the entire array. Instead, being a pointer, the sizeof() would return the size of the pointer (4 bytes on 32-bit machines)! Also, when we pass arrays to functions, we actually pass a pointer. Once again, we should not use the sizeof() in the called function since the array is actually as a pointer. For such cases, we should explicitly pass the length of the array as a function parameter.

Pointers and Strings

Since C stores strings as array of characters, we can also use pointers to access individual characters in a string. Let us use an example of copying a string to demonstrate accessing elements of a character array (a string, actually) by merely incrementing the pointers.

In the example provided below, the char pointers ptr_name and ptr_copy_name point to the two char arrays. Using a while loop, we assign values from ptr_name to ptr_copy_name, one character at a time, and then increment both pointers to point to next character. Since, the pointers are of type "char *", every time we increment these counters, the new address advances by size of one char. We terminate the while loop when the value of character pointed by ptr_name become the NUL terminator character ('\0').

```
#include <stdio.h>

int main () {
    char name[25] = "Leonardo da Vinci";
    char copy_name[25] = "Van Gogh";
    char *ptr_name = NULL;
    char *ptr_copy_name = NULL;

    ptr_name = name;
    ptr_copy_name = copy_name;

    printf("Before copying: \n");
    printf("ptr_name: %20s ptr_copy_name: %15s\n", ptr_name, ptr_copy_name);
    printf("    name: %20s     copy_name: %15s\n", name, copy_name);

    while ( (*(ptr_copy_name++) = *(ptr_name++)) != '\0') ;

    printf("\nAfter copying: \n");
    printf("ptr_name: %20s ptr_copy_name: %15s\n", ptr_name, ptr_copy_name);
    printf("    name: %20s     copy_name: %15s\n", name, copy_name);
    return 0;
}
```

The output (provided below) shows that once we make the pointer point to a string and print it, the output is identical whether we print the string itself or the pointer. Lastly, after copying, we find that both ptr_name and ptr_copy_name are NULL -- this is because at the the end copying, both strings have advanced till the end of ptr_name and hence, they both now point to '\0'.

```
Before copying:
ptr_name:    Leonardo da Vinci ptr_copy_name:        Van Gogh
    name:    Leonardo da Vinci     copy_name:        Van Gogh
```

```
After copying:
ptr_name:                       ptr_copy_name:
    name:    Leonardo da Vinci     copy_name: Leonardo da Vinci
```

In the above example, since the while clause is executed first and the check for the NUL character is done later, the example also copies the last NUL character. We should also note that the copy_name string has enough storage (25 bytes) to accommodate the "Leonardo da Vinci" string. If it were to have less storage, then we would end up over-running some of the buffer. As a general rule, when dealing with strings, it is important to pay extra attention to the storage size and the NUL character.

Creating Arrays using malloc()

Being able to define an array with a given size is nice, but sometimes we do not know the array size in advance. For such cases, a declaration of array specifying its initial size, this approach would not work. Instead, we can use a malloc() call to allocate the array on the fly. We can use malloc() to allocate both one-dimensional and two-dimensional arrays. As with every malloc() calls, we need to make sure that we free() the memory once we are done!

With that, let us look at a handful of examples that use malloc() calls to create one-dimensional and two-dimensional arrays respectively.

The first example uses malloc to allocate a one-dimensional array. It starts by defining an array as "int* painting_array"; this works since arrays can be also be referenced as pointers. The example then assigns values to array elements and prints them. In the end, it frees the allocated memory using the free() call.

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_OF_COLUMNS 4

int main () {
    int i;
    int* painting_array;

    /* Allocate the array */
    painting_array = (int *)malloc(sizeof(int) * NUM_OF_COLUMNS);
    if (!painting_array) {
        fprintf(stderr, "Malloc failed \n");
        return -1;
    }

    /* Assign values to array elements */
    painting_array[0] = 1000;
    painting_array[1] = 1001;
    painting_array[2] = 1002;
    painting_array[3] = 1003;

    for (i = 0; i < NUM_OF_COLUMNS; i++) {
```

```
        printf("[i: %d] painting id: %d (Address: %p)\n",
            i, painting_array[i], &painting_array[i]);
    }

    /* Free the allocated array */
    free(painting_array);
    return 0;
}
```

As expected, the output prints all the elements of the array. Also, looking at the addresses, elements of the array form a contiguous block of memory. If you notice something unusual about these addresses (they are not exactly 4 bytes), then that is okay since we cannot make any assumption about the address of the block returned from a malloc call.

```
[i: 0] painting id: 1000 (Address: 0x90a3008)
[i: 1] painting id: 1001 (Address: 0x90a300c)
[i: 2] painting id: 1002 (Address: 0x90a3010)
[i: 3] painting id: 1003 (Address: 0x90a3014)
```

Our second example uses malloc to allocate a two-dimensional array. It starts by defining an array as "int** painting_array" -- this means that it is a pointer to a pointer. How does that translate to an two-dimensional array? Well, the first pointer defines all the rows of the array and each row is identified by yet another pointer. This works since a two-dimensional array is nothing but an array of one-dimensional arrays.

The example then allocates a pointer that represents the rows of the painting_array. Next, it uses malloc to build individual elements of the painting_array row. In the end, it frees the allocated memory using the free() call.

Note the ordering of the free() calls is opposite to that of the malloc() calls. We allocate the painting_array first and then allocate the rows. With free(), we first free the rows, and then free the painting_array. Needless to say, if we free the painting_array pointer first, then would loose the handle to all rows and we would end up with a memory leak!

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_OF_ROWS 2
#define NUM_OF_COLUMNS 4

int main () {
    int i, j;
    int **painting_array;

    /* Let us use malloc to create rows first */
    painting_array = (int** )malloc(sizeof(int*) * NUM_OF_ROWS);
    if (!painting_array) {
        fprintf(stderr, "Malloc failed\n");
        goto AllDone;
    }
    /* Next, let us allocate one-dimensional arrays (rows) for each column
*/
```

```
    for (i=0; i < NUM_OF_ROWS; i++) {
        painting_array[i] = (int *)malloc(sizeof(int) * NUM_OF_COLUMNS);
        if (!painting_array[i]) {
            fprintf(stderr, "Malloc failed\n");
            goto AllDone;
        }
    }

    /* Painting Unique IDs */
    painting_array[0][0] = 1000;
    painting_array[0][1] = 1001;
    painting_array[0][2] = 1002;
    painting_array[0][3] = 1003;

    /* Number of paintings available in the store */
    painting_array[1][0] = 100;
    painting_array[1][1] = 10;
    painting_array[1][2] = 10;
    painting_array[1][3] = 11;

    for (i=0; i < NUM_OF_COLUMNS; i++) {
        printf("[i: %d] painting id (%p): %d Availability (%p): %d\n",
            i, &painting_array[0][i], painting_array[0][i],
            &painting_array[1][i], painting_array[1][i]);
    }

AllDone:
    /* Free the allocated columns for each row */
    for (i=0; i < NUM_OF_ROWS; i++) {
        if (painting_array[i]) {
            free(painting_array[i]);
        }
    }
    if (painting_array) {
        free(painting_array);
    }
    return 0;
}
```

With the output, note that each of the row use a contiguous piece of memory. This is understandable since we use a separate malloc call for each of the two rows.

```
[i: 0] painting id (0x8528018): 1000 Availability (0x8528030): 100
[i: 1] painting id (0x852801c): 1001 Availability (0x8528034): 10
[i: 2] painting id (0x8528020): 1002 Availability (0x8528038): 10
[i: 3] painting id (0x8528024): 1003 Availability (0x852803c): 11
```

Lastly, when C defines a multi-dimensional arrays, it allocates a contiguous chunk of memory such that it can accommodate all the elements. Thus, if we have a 2 dimensional array with k rows and m columns, then the address of each row would be continuous and the address of the element starting the next row would be immediately after the address of the last element in the previous row.

Before we wrap up this page, let us confirm this using a simple example. The example defines a 2-dimensional array and then prints the address of each element row by row. Here it is:

```
#include <stdio.h>

#define NUM_OF_ROWS 2
#define NUM_OF_COLUMNS 4

int main () {
    int painting_array[NUM_OF_ROWS][NUM_OF_COLUMNS];
    int i, j;

    /* Let us print the addresses of elements */
    for (i=0; i < NUM_OF_ROWS; i++) {
        for (j=0; j < NUM_OF_COLUMNS; j++) {
            printf("\t %p ", &painting_array[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

And, here is the output:

```
        0xbffb6ae8      0xbffb6aec      0xbffb6af0      0xbffb6af4
        0xbffb6af8      0xbffb6afc      0xbffb6b00      0xbffb6b04
```

Pointers in C: Passing Pointers as Function Parameters

When we call a function in C, typically, we pass a set of parameters to that function. With respect to pointers, there are three ways in which a caller can can pass parameters to the functions: (1) by value, (2) by reference, and, (3) by reference of the reference. These methods serve different purposes and it is important to understand them in detail.

Passing by Value

When we pass a parameter by value, then we pass a copy of the variable to the called function.

Let us go through an example (provided below) to understand this better. Here, the main() function calls pass_variable_param() and instead of passing actual x, it passes a copy of x. In this example, var_temp is nothing but a copy of x.

```
#include <stdio.h>

void pass_variable_param (int var_temp) {
    printf("\t[%s] Passed value is %d (address: %p)\n",
            __FUNCTION__, var_temp, &var_temp);
    var_temp *= 10;
    printf("\t[%s] Updated value is %d (address: %p)\n",
            __FUNCTION__, var_temp, &var_temp);
}
```

```
int main () {
    int x = 100;

    printf("[%s] Before function call, value is %d (address: %p)\n",
            __FUNCTION__, x, &x);
    pass_variable_param(x);
    printf("[%s] After function call, value is %d (address: %p)\n",
            __FUNCTION__, x, &x);
}
```

The output (provided below) shows that even though we change the value of var_temp in the function pass_variable_param(), the value of x in main() remains unchanged. This is the expected behavior, since after all, we are passing a copy of x and changes to a copy would not be reflected in variable x. Another proof of this behavior is that x and its copy (var_temp) have different addresses.

```
[main] Before function call, value is 100 (address: 0xbfc9b4ec)
       [pass_variable_param] Passed value is 100 (address: 0xbfc9b4d0)
       [pass_variable_param] Updated value is 1000 (address: 0xbfc9b4d0)
[main] After function call, value is 100 (address: 0xbfc9b4ec)
```

Passing by Reference

When we pass reference (aka address) of a variable instead of its value, then we are actually passing the address of the variable. Thus, the called function can manipulate the value stored at the address instead of manipulating a copy of the variable defined in the calling function. This approach allows the change made in the called function to become visible in the caller function as well.

Therefore, if we wish that the changes made in the called function become visible in the caller function, then we need to pass by reference instead of passing by value.

With that, let us rewrite the earlier example such that the main() function passes the variable by reference instead of by value. This allows the called function (pass_pointer_param()) to directly manipulate the value stored at the address of x, which means x itself.

```
#include <stdio.h>

void pass_pointer_param (int *ptr_temp) {
    printf("\t[%s] Passed value is %d (address: %p)\n",
        __FUNCTION__, *ptr_temp, ptr_temp);
    *ptr_temp *= 10;
    printf("\t[%s] Updated value is %d (address: %p)\n",
        __FUNCTION__, *ptr_temp, ptr_temp);
}

int main () {
    int ret_val, x = 100;

    printf("[%s] Before function call, value is %d (address: %p)\n",
```

```
        __FUNCTION__, x, &x);
    pass_pointer_param(&x);
    printf("[%s] After function call, value is %d (address: %p)\n",
        __FUNCTION__, x, &x);
    return 0;
}
```

When we see the output, we find that the changed value gets reflected in the main() function as well. Also, both functions see the same address for the variable.

```
[main] Before function call, value is 100 (address: 0xbfe585dc)
       [pass_pointer_param] Passed value is 100 (address: 0xbfe585dc)
       [pass_pointer_param] Updated value is 1000 (address: 0xbfe585dc)
[main] After function call, value is 1000 (address: 0xbfe585dc)
```

Before we move on, we would like to reflect on a few miscellaneous but related behavior of using pointers as functions params.

First, this method is especially useful when we pass a variable that has a much bigger storage as compared to a simple integer in the above code; an example of such storage could be a data structure that has a large size. Passing a large variable by value can be inefficient since we need to make a copy of the variable before passing it.

Second, the same logic applies when the function returns a value. If we are dealing with a large data structure, then without passing by reference, the function would end up passing by value, which would mean making another copy.

Third, functions have an inherent limitation of being able to return a single value. Using references, we can pass multiple variables and the function can update values of these variables within the function. In this way, we can break the inherent limitation of functions returning a single value.

Lastly, it would be a good place to point out that when we pass an array to a function, then C passes them by reference instead of value. If you have ever wondered why sizeof() does not give the correct size of the array in a called function, then this is the reason! The called function only gets a pointer and so calling a sizeof() on the passed array would simply return the size of the pointer. Needless to say, passing by reference is an efficient design since a C array can contain a large number of elements and passing them by value would mean making a copy of all those values with each call.

Passing Reference of the Reference

This approach is useful when one has to manipulate the address of the passed pointer itself in the called function; in such a case, passing by reference would not work.

In case you are wondering, what can be a possible use-case of passing a reference of reference, then, here is one. Let us say, we have a requirement for manipulating a pointer in the called function -- there can be cases, when we do a malloc() inside the called function and not in the

caller function. For example, we can pass a pointer and make it point to a data structure allocated in the called function. Once the execution of the called function is complete, then we would like to get the handle of the new pointer instead of the earlier passed pointer; the earlier pointer might as well be pointing to NULL!

Let us use our earlier example to demonstrate the need for passing a reference of a reference. We start by providing an example that incorrectly attempts to use the address of a malloced pointer from outside the called function.

```
#include <stdio.h>
#include <stdlib.h>

void pass_pointer_param_wrong (int *ptr_temp) {
    printf("\t[%s] Passed pointer address is %p\n",
            __FUNCTION__, ptr_temp);
    ptr_temp = (int *)malloc(sizeof(int));
    printf("\t[%s] Updated pointer address is %p\n",
            __FUNCTION__, ptr_temp);
}

int main () {
    int *ptr_int = NULL;

    printf("[%s] Before function call, pointer address is %p\n",
        __FUNCTION__, ptr_int);
    pass_pointer_param_wrong(ptr_int);
    printf("[%s] After function call, pointer address is %p\n",
        __FUNCTION__, ptr_int);
    return 0;
}
```

When we do a malloc in pass_pointer_param_wrong(), ptr_temp points to a new address. However, since we are passing only the address of the pointer variable, a change in the address of the pointer in pass_pointer_param_wrong() does not get reflected in the calling function, main(). The output of this program demonstrates this failure:

```
[main] Before function call, pointer address is (nil)
       [pass_pointer_param_wrong] Passed pointer address is (nil)
       [pass_pointer_param_wrong] Updated pointer address is 0x945e008
[main] After function call, pointer address is (nil)
```

In the above example, the malloc() call changes the address pointed to by the pointer from (nil) to 0x9890008. However, when the call returns back to main(), the information of the new address is lost. This is not all! What is worse is that because the main() function does not have a pointer to the malloced data and so it cannot call free() -- there are very few programming mistakes that are as grave as not freeing an allocated memory!

To avoid this, we need to pass a double pointer as a parameter. We provide the modified code. Not only does this method correctly reflects the new address in the main() function, it also frees the allocated memory!

```
#include <stdio.h>
#include <stdlib.h>

void pass_pointer_param_correct (int **ptr_temp) {
    printf("\t[%s] Passed value of the double pointer is %p\n",
        __FUNCTION__, *ptr_temp);
    *ptr_temp = (int *) malloc(sizeof(int));
    printf("\t[%s] Updated value of the double pointer is %p\n",
        __FUNCTION__, *ptr_temp);
}

int main () {
    int *ptr_int = NULL;
    int **ptr_ptr_int = &ptr_int;

    ptr_ptr_int = &ptr_int;
    printf("[%s] Before function call, pointer address is %p\n",
        __FUNCTION__, ptr_int);
    printf("[%s] Before function call, address of double-pointer is %p\n",
        __FUNCTION__, ptr_ptr_int);

    pass_pointer_param_correct(ptr_ptr_int);
    ptr_int = *ptr_ptr_int;

    printf("[%s] After function call, pointer address is %p\n",
        __FUNCTION__, ptr_int);
    printf("[%s] After function call, address of double-pointer is %p\n",
        __FUNCTION__, ptr_ptr_int);
    free(ptr_int);
    return 0;
}
```

Here is the program output:

```
 [main] Before function call, pointer address is (nil)
 [main] Before function call, address of double-pointer is 0xbf95e3e8
       [pass_pointer_param_correct] Passed value of the double pointer is
(nil)
       [pass_pointer_param_correct] Updated value of the double pointer is
0x8b9d008
 [main] After function call, pointer address is 0x8b9d008
 [main] After function call, address of double-pointer is 0xbf95e3e8
```

Pointers in C: Pointers and Function Names

Since pointers are used to locate content at a given address, pointers can also be made to point to the address of a functions since functions have addresses as well. In other words, we can use pointers to pass name of functions. This method can be useful for cases, where we have a set of functions and we need to decide the right function during the run-time.

You might be wondering what happens when we have inline functions since inline functions do not have addresses. Would function pointers still work? Well, the good news is that a compiler

would create an out-of-line instance of the inline function, if it sees that there is a function pointer pointing to it. Long story short, it would still work.

As usual, let us write a simple example to put our understanding to test. We provide below an example that uses a pointer to point to various functions.

```
#include <stdio.h>

int subtract(int x, int y) {
    return (x-y);
}

int add(int x, int y) {
    return (x+y);
}

int multiply(int x, int y) {
    return (x*y);
}

int divide(int x, int y) {
    if ( y != 0)
        return (x/y);
    else
        printf("Division by zero is not allowed\n");
    return 0;
}

int main () {
    int ((*func)(int, int));
    int a = 100;
    int b = 10;

    func = add;
    printf("Addition Result is %d\n", func(a,b));

    func = subtract;
    printf("Subtraction Result is %d\n", func(a,b));

    func = multiply;
    printf("Multiplication Result is %d\n", func(a,b));

    func = divide;
    printf("Division Result is %d\n", func(a,b));
    return 0;
}
```

In the above example, pointer func is a function pointer and the list of arguments taken by the function is defined. Please pay close attention to the declaration of function pointer -- the function pointer is defined such that it retains its signature, "((*func)(int,int)" and the starting "int" means that it returns an int. A function pointer can point to a function if and only if the signature of the function is same as that of the function pointer.

We use func to point to add(), subtract(), multiply(), and divide() functions. These functions have the same argument signature, and so the pointer func can point to any one of them. However, if a function were to have different a argument signature, then the func pointer cannot point to that function.

Here is the output, when we run the above program:

```
Addition Result is 110
Subtraction Result is 90
Multiplication Result is 1000
Division Result is 10
```

In fact, if we wanted to do too much, then we can even define an array of function pointers. The advantage here would be that, we do not have to point the pointer func again and again to different functions. We initialize the function array at the beginning to appropriate functions and we should be all set!

Let us modify the earlier program to use a function array. To avoid duplication, we don't keep the definition of functions, add, subtract, multiply, and divide in this example. Notice the definition of the function pointer array, func, such that it now contains LEN_BINARY_OPERATIONS to specify the array size. The output for this code is same as before, so we omit it.

```
#include <stdio.h>

enum binary_operations {
    ADD,
    SUBTRACT,
    MULTIPLY,
    DIVIDE,
    LEN_BINARY_OPERATIONS,
};

void main () {
    int ((*func[LEN_BINARY_OPERATIONS])(int, int));
    int a = 100;
    int b = 10;

    func[ADD] = add;
    func[SUBTRACT] = subtract;
    func[MULTIPLY] = multiply;
    func[DIVIDE] = divide;

    printf("Addition Result is %d\n", (func[ADD])(a,b));
    printf("Subtraction Result is %d\n", func[SUBTRACT](a,b));
    printf("Multiplication Result is %d\n", func[MULTIPLY](a,b));
    printf("Division Result is %d\n", func[DIVIDE](a,b));
    return 0;
}
```

Accessing functions through pointers may sound awesome (and in many cases, it is!), but overusing it can seriously ruin code-readability. This is because, we can have a pointer pointing

to one function at one time and yet another function at another time. All of this would make the reader focus more on the flow of function pointers instead of the actual application logic, and likely leave him confused. So, it is one of those powers that we should use as little as possible. When in doubt, avoid using function pointers -- it would go a long way in making your code readable and would do everyone good!

Data Structures: Introduction

C defines several built-in storage data types like char, short, int, long, array, string etc. In fact, it does not stop there -- it allows us to even build data structures that can keep custom data types. Data structures can combine various data types (like int, char, etc) to create a more customized encapsulation that is tailored for the application requirement. Data structures are versatile and can include other data structures as well. It is this ability to bind together storage of various types that data structures play an important role in C programming.

We can use the "struct" keyword to define a structure. Following this keyword, we need to provide the name of the structure. Next, we can provide data of various types, each separated by a semi-colon, and enclose them within braces. Each of these individual data types are referred to as data structure members.

Let us say, we want to write an application that handles an inventory of paintings and would like to use a data structure to hold information relevant to a painting. We can define a structure as follows; do not forget the semi-colon at the end of the structure definition.

```
struct painting_frame {
    int painting_id;
    int width;
    int height;
    float price;
    char painter[50];
};
```

The above structure contains data members (or fields) that store painter's name, price of the framed painting, along with the width and height of the frame; the structure also stores a unique painting ID for each painting. Thus, we can have an instance of this data structure for each painting and a common data representation for all paintings.

We do not have to stop here! We can even define an array of structures. The code snippet provided below, defines all_frames as an array of painting_frame data structures. The value of MAX_FRAMES is 1000 and that means all_frames can store information about 1000 paintings.

```
#define MAX_FRAMES 1000
struct painting_frame all_frames[MAX_FRAMES];
```

Once we have defined a data structure, we need to access its members. C allows us to refer to members of a data structure using the dot (".") operator.

Let us see an example that illustrates the usage of data structure. The example (provided below) initializes a single structure as well as an array of the same structure and also accesses their members. To keep the code compact, we provide a function, print_painting_frame(), that prints various members of the painting_frame data structure.

```
#include <stdio.h>
#include <string.h>

#define MAX_FRAMES 5

typedef struct painting_frame {
    int painting_id;
    int width;
    int height;
    float price;
    char painter[50];
} painting_frame_t;

void print_painting_frame(painting_frame_t *frame) {
    printf("ID: %d Width: %d Height: %d Painter: %s\n",
            (*frame).painting_id, (*frame).width,
            (*frame).height, (*frame).painter);
}

int main() {
    painting_frame_t sample_frame;              /* Single data structure */
    painting_frame_t all_frames[MAX_FRAMES];    /* An array of data
structures */
    int i;

    /* Initialize members of sample_frame */
    sample_frame.painting_id = 100;
    sample_frame.width = 50;
    sample_frame.height = 100;
    memcpy(sample_frame.painter,
          "Leonardo da Vinci", strlen("Leonardo da Vinci") + 1);

    printf("Let us print info about sample_frame\n");
    print_painting_frame(&sample_frame);

    /* Initialize members of an array of painting_frame */
    for (i=0; i < MAX_FRAMES; i++) {
        all_frames[i].painting_id = i;
        all_frames[i].width = 50;
        all_frames[i].height = 100;
        /* Let us say, these paintings are painted by Van Gogh */
        memcpy(all_frames[i].painter, "Van Gogh", strlen("Van Gogh") + 1);
    }

    printf("\nLet us print info about painting array\n");
    for (i=0; i < MAX_FRAMES; i++) {
        print_painting_frame(&all_frames[i]);
    }
    return 0;
}
```

Here is the output of the above code:

```
Let us print info about sample_frame
ID: 100 Width: 50 Height: 100 Painter: Leonardo da Vinci

Let us print info about painting array
ID: 0 Width: 50 Height: 100 Painter: Van Gogh
ID: 1 Width: 50 Height: 100 Painter: Van Gogh
ID: 2 Width: 50 Height: 100 Painter: Van Gogh
ID: 3 Width: 50 Height: 100 Painter: Van Gogh
ID: 4 Width: 50 Height: 100 Painter: Van Gogh
```

Note that it is possible to initialize the value of various members of a structure during the declaration itself. The following code defines a simple structure and also initializes its members during the definition of temp_painting_id.

```
typedef struct painting_dimensions {
    float length;
    float width;
    float height;
} painting_d;

painting_d temp_painting_d = {100.0, 80.0, 10.0};
```

Lastly, data structures can include other data structures. For example, if we were to define a structure to hold various information of bidding of a frame along with the painting_frame_t itself, then here is how we can define it:

```
typedef struct frame_bid {
    painting_frame_t painting_frame;
    int bid_value;
    char *bidder_name;
} frame_bid_t;
```

Data structures are like any other variables and so even though, they are a collection of variables, one can still assign a value of one data structure to another. The following example uses a painting_frame_t structure to define two variables, frame1 and frame2. Then it assigns frame1 to frame2. When we run the example, we find that frame2 has same value as frame1.

```
#include <stdio.h>

typedef struct painting_frame {
    int painting_id;
    int width;
    int height;
} painting_frame_t;

void print_painting_frame(painting_frame_t *frame) {
    printf("ID: %d Width: %d Height: %d\n",
            (*frame).painting_id,
            (*frame).width,
            (*frame).height);
}
```

```
int main() {
    painting_frame_t frame1 = {100, 80.0, 10.0};
    painting_frame_t frame2;

    frame2 = frame1;
    print_painting_frame(&frame1);
    print_painting_frame(&frame2);
    return 0;
}
```

Data Structures as Pointers

Since data structures can potentially store large amounts of data, we often access data structures using pointers; this is especially true when we pass data structures to functions. Passing by pointer ensures that only the address of the structure is passed instead of a copy of the data contained by the structure; passing by value can be inefficient.

Unlike the dot operator, when referring to a member of a data structure via pointer, we need to use the structure pointer operator, "->", in place of the dot operator. Trying to access a data member using the dot operator (as "new_frame.painting_id") for a pointer would be a compilation error.

```
painting_frame_t sample_frame;
painting_frame_t *new_frame = &sample_frame;

//Change the ID to new ID
new_frame->painting_id = 101;
```

Of course, the other alternative would be to call "(*new_frame).painting_id", where we access the value of the pointer first and then use the dot operator. Clearly, using "->" is more convenient when dealing with pointers!

Next, we provide a sample address of a data structure and a pointer pointing to it. In the figure provided below, new_frame is a pointer that points to the address of sample_frame data structure. Like the case of other pointers, the value of the new_frame is simply the address of the sample_frame data structure.

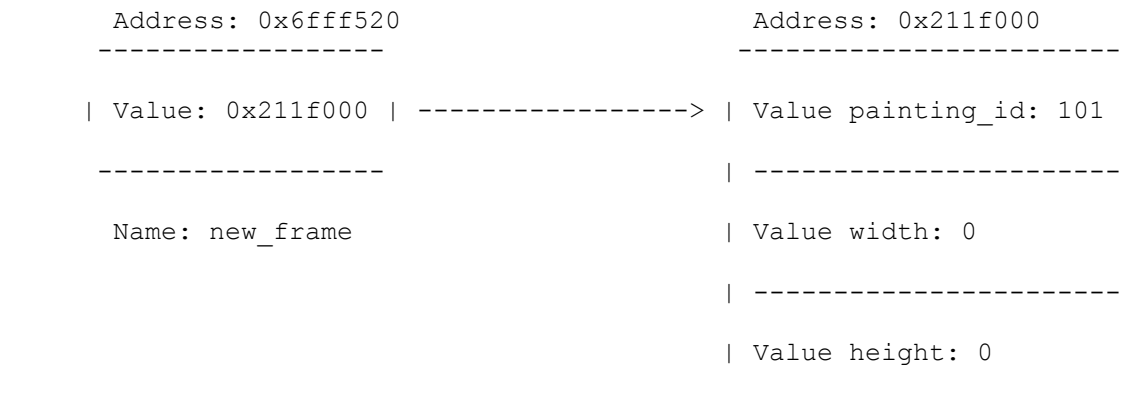


Figure: A pointer stores an address of a data structure

We do not necessarily have to define a structure as a variable and then use a pointer to refer to it. We can define a pointer from the very beginning and then allocate memory to it, and then continue to use it. Do not forget to free the allocated memory, once you are done!

Here is a sample program that modifies the earlier program to make sample_frame a pointer from the get-go!

```
 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>

 typedef struct painting_frame {
     int painting_id;
     int width;
     int height;
     float price;
     char painter[50];
 } painting_frame_t;

 void print_painting_frame(painting_frame_t *frame) {
     printf("ID: %d Width: %d Height: %d Painter: %s\n",
             frame->painting_id, frame->width,
             frame->height, frame->painter);
 }

 int main() {
     painting_frame_t *sample_frame;

     sample_frame = (painting_frame_t *)malloc(sizeof(painting_frame_t));
     if (sample_frame == NULL) {
         printf("Error: Malloc Failed\n");
         return -1;
     }

     /* Initialize members of sample_frame */
     sample_frame->painting_id = 100;
     sample_frame->width = 50;
     sample_frame->height = 100;
     memcpy(sample_frame->painter, "Leonardo da Vinci", strlen("Leonardo da
Vinci") + 1);
```

```
    printf("Let us print info about sample_frame\n");
    print_painting_frame(sample_frame);

    free(sample_frame);
    return 0;
}
```

Here is the output:

```
Let us print info about sample_frame
ID: 100 Width: 50 Height: 100 Painter: Leonardo da Vinci
```

Unions

Like structures, unions are another constructs that allow us to store multiple values (aka data members). The difference between the two is that with unions, we can hold only one value at a given time. In that sense, all the values stored in a union share the same common space. The size of the common space of a union is the size of the largest member of the union.

We define a union using the "union" keyword. Union members can be accessed just like that of data structures. If we are using a union variable, then we can use the dot operator ("."). If we are using a union pointer, then we can use the pointer operator ("->").

Since a union can hold different values at different times, one way to avoid ambiguity is to maintain a state for each union instance. Whenever, we update the value of the union to a new type, we need to update the state as well.

With that, let us see an example that uses a union to hold various values for tax types for a painting. The example uses a union painting_tax that contains various storage types reflecting taxes and commissions. Next, it also uses an enum of tax_type to indicate each of the types. The example also prints address of various union members.

```
#include <stdio.h>
#include <string.h>

typedef union painting_tax {
    char tax_code[8];
    double commission;
    int city_tax;
} painting_tax_t;

typedef enum tax_type {
    TAX_CODE,
    COMMISSION,
    CITY_TAX
} tax_type_t;

void print_tax (painting_tax_t *tax, tax_type_t type) {
    switch (type) {
        case TAX_CODE:
        printf("The Tax code is %s (Address: %p)\n",
```

```
                tax->tax_code, &tax->tax_code);
        break;

        case COMMISSION:
        printf("The Commission is %f (Address: %p)\n",
                tax->commission, &tax->commission);
        break;

        case CITY_TAX:
        printf("The City tax is %d (Address: %p)\n",
                tax->city_tax, &tax->city_tax);
        break;

        default:
        printf("Error: unknown category\n");
        break;
    }
}

int main () {
    painting_tax_t tax;
    tax_type_t type;

    printf("The sizeof painting_tax_t is %d (Address: %p)\n",
            sizeof(tax), &tax);

    type = TAX_CODE;
    strcpy(tax.tax_code, "TXC-101");
    print_tax(&tax, type);

    type = COMMISSION;
    tax.commission = 9.25;
    print_tax(&tax, type);

    type = CITY_TAX;
    tax.city_tax = 10;
    print_tax(&tax, type);

    /* It is an error to not update the type variable */
    strcpy(tax.tax_code, "TXC-101");
    print_tax(&tax, type);

    return 0;
}
```

The output demonstrates that each time we call print_tax(), it prints the corresponding value as dictated by the variable type. Also, for the last case, when we update the value of tax_code but do not update the type to TAX_CODE, print_tax() incorrectly prints the value of tax_code.

```
The sizeof painting_tax_t is 8 (Address: 0xbff16734)
The Tax code is TXC-101 (Address: 0xbff16734)
The Commission is 9.250000 (Address: 0xbff16734)
The City tax is 10 (Address: 0xbff16734)
The City tax is 759388244 (Address: 0xbff16734)
```

The output also shows that the size of the union is 8, which equals the size of the tax_code or the commission. Lastly, when we print the address of the union or its values, they all point to the same memory location.

Data Structures: Alignment

On most platforms, memory is read in words and so accessing data structures becomes faster if they are aligned along a word boundary. The word size refers to the size of a processor register. On a 32 bit processor, the word size is 32 bits and on a 64 bit processor, it is 64 bits.

What this also means is that if we define members of a data structure such that they are not aligned as words, then the compilers might sneak in a few padding bytes so that they becomes word-aligned. Trying to access data structure members that are not aligned means the compiler will have to generate more code and that will likely make your application slower. So, in most cases, what compiler is doing is actually the right thing.

Let us understand this using an example. The following structure contains a char array of 2 bytes, two integer values and a short. Thus, the total storage on a 32-bit machine should be 12 bytes. However, if we were to do a sizeof() for this structure, then the value would actually be 16!

```
typedef struct painting_frame {
    char color_code[2];
    int painting_id;
    short width;
    int height;
} painting_frame_t;
```

So, why is the actual size of the structure different from the sum of sizes of its members. Well, this is because the members of this structure are not aligned and hence, the extra bytes are added by the compiler. The definition starts with a field color_code which is a char array of 2 chars. After that, we add an integer painting_id. Since we cannot fit the char array and an integer in the same word (2 + 4 would be 6 byes, which would go beyond a word), the compiler adds a padding of 2 bytes after the color_code and starts painting_id from the next row! After that, we have the field width of type short followed by field height of type integer. And once again, the compiler would add a padding of 2 bytes to the width field.

You could argue that is this so bad? The short answer is that it is. The actual size of all the data structure members is: 2 + 2 + 4 + 4 or 12 bytes. Thus, when the compiler bloats it to 16, the overhead is roughly 33.33%. If the application uses a large amount (an array say) of such data structures, then we would waste 33.33% of all the memory allocated for painting_frame_t. Not exactly efficient.

One way to avoid this overhead is to see if we can shuffle members of the data structure so that they can fit better in a word. If we were to move the width member immediately after color_code, then these two field (color_code and width) can fit in one word! The remaining two members, painting_id and height can have their own words. With that, the size of the data structure would becomes 4 + 4 + 4 or 12 bytes.

```
typedef struct painting_frame {
    char color_code[2];
    short width;
    int painting_id;
    int height;
} painting_frame_t;
```

Let us demonstrate this using an example that uses both aligned and non-aligned versions of the structure. This example also uses offsetof() function that returns the offset of a data structure member from the start of the data structure. The offsetof() function is published by the "stddef.h" header file.

```
#include <stdio.h>
#include <stddef.h>

typedef struct painting_frame_unaligned {
    char color_code[2];
    int painting_id;
    short width;
    int height;
} frame_unaligned_t;

typedef struct painting_frame_aligned {
    char color_code[2];
    short width;
    int painting_id;
    int height;
} frame_aligned_t;

int main() {
    printf("Printing values for frame_unaligned:\n");
    printf("Sizeof: %d\n", sizeof(frame_unaligned_t));
    printf("Offset of color_code: %d\n", offsetof(frame_unaligned_t,
color_code));
    printf("Offset of painting_id: %d\n", offsetof(frame_unaligned_t,
painting_id));
    printf("Offset of width: %d\n", offsetof(frame_unaligned_t, width));
    printf("Offset of height: %d\n", offsetof(frame_unaligned_t, height));

    printf("\nPrinting values for frame_aligned:\n");
    printf("Sizeof: %d\n", sizeof(frame_aligned_t));
    printf("Offset of color_code: %d\n", offsetof(frame_aligned_t,
color_code));
    printf("Offset of width: %d\n", offsetof(frame_aligned_t, width));
    printf("Offset of painting_id: %d\n", offsetof(frame_aligned_t,
painting_id));
    printf("Offset of height: %d\n", offsetof(frame_aligned_t, height));

    return 0;
}
```

The output confirms our fear! For the first unaligned case, the size is indeed 16 bytes. This is also reflected in the return value of offsetof() for painting_id. The lesson here is that if there is a

possibility that we can redefine a structure by shuffling the fields so that they align better, then by all means, we should!

```
Printing values for frame_unaligned:
Sizeof: 16
Offset of color_code: 0
Offset of painting_id: 4
Offset of width: 8
Offset of height: 12

Printing values for frame_aligned:
Sizeof: 12
Offset of color_code: 0
Offset of width: 2
Offset of painting_id: 4
Offset of height: 8
```

Yet another way (and rather risky way) to avoid the padding overhead is to force the compiler to not align fields -- we can do this using the packed attribute provided by GCC. Here is how, we can use this attribute on the painting_frame_unaligned data structure.

```
#include <stdio.h>
#include <stddef.h>

typedef struct __attribute__((__packed__)) painting_frame_unaligned {
    char color_code[2];
    int painting_id;
    short width;
    int height;
} frame_unaligned_t;

int main() {
    printf("Printing values for frame_unaligned:\n");
    printf("Sizeof: %d\n", sizeof(frame_unaligned_t));
    printf("Offset of color_code: %d\n", offsetof(frame_unaligned_t,
color_code));
    printf("Offset of painting_id: %d\n", offsetof(frame_unaligned_t,
painting_id));
    printf("Offset of width: %d\n", offsetof(frame_unaligned_t, width));
    printf("Offset of height: %d\n", offsetof(frame_unaligned_t, height));

    return 0;
}
```

The output shows that the packed attribute does indeed provide a more compact binding. However, this comes at a cost. Since the compilers must generate code to read unaligned members, this will slow down your program. And, if frame_unaligned_t is being used heavily, then this would bring noticeable latency in your application. So, the first thing to avoid memory overhead coming from data structure misalignment is to check if we can manually shuffle the data structure members -- this way, we can avoid the performance penalty paid when using the packed attribute.

```
Printing values for frame_unaligned:
```

```
Sizeof: 12
Offset of color_code: 0
Offset of painting_id: 2
Offset of width: 6
Offset of height: 8
```

One last note. We should avoid accessing members of a data structure by navigating the memory allocated to a data structure. Clearly, padding bytes has an element of surprise and using raw memory offset may fail since we may not know in advance how many bytes are added. For the case, when we use the packed attribute, it is possible that the value might not be aligned well -- and on some systems, this may generate an alignment fault. To conclude, as long as we stick with using the dot operator (".") or the pointer operator ("->"), we should be covered.

C String Library: Copy Functions

C provides a set of library functions for manipulating both string and non-string data. These functions are published using the "string.h" header file and we should include this header file to use them. A string data uses a pointer to char ("char *") and a non-string data uses a pointer to void ("void *").

Before we go any further, it would do us good to recall that C represents strings as an array of char types. Each char type requires one byte of storage. Thus, a string is an array of one byte-sized elements. When C stores a string in an array, it uses '\0' (NUL termination character) as the last character to mark the end of the string. The NUL character is a critical aspect of C string, because without that, C would not be able to know the end of the string.

In this section, we focus on functions that copy data (string or non-string) from one location to another.

Overview of Functions

Let us begin by providing the copying functions; these functions help us copy both string and non-string data.

```
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
void *memcpy(void *dest, void *src, size_t n);
void *memmove(void *dest, void *src, size_t n);
```

Function strcpy() copies a source string (src) into a destination string (dest), including the NUL character present at the end of the source string. The function strncpy() also does the same thing, but it only copies a maximum of "n" bytes. If strncpy() reaches NUL character (that marks the end of the src string) before reaching "n" bytes, then the copying terminates there. When passing "n", we should ensure that it includes the NUL character of the src string as well.

For both cases, it is equally important that the dest string should have enough buffer to accommodate the src string being copied along with the NUL character; thus, the length should be at least equal to the length of the source string plus the NUL character.

For strcpy() and strncpy() functions, if there is an overlap in the source and destination strings, then the behavior is undefined; for strncpy(), the overlap would occur if there is an overlap between the first n bytes of the source string and the first n bytes of the destination string.

The next function, memcpy(), allows us to copy "n" bytes from src buffer to the dest buffer. Since memcpy() already takes "n" size_t as input, it is in fact closely related to strncpy and not strcpy. Unfortunately, similar to strcpy() and strncpy(), if the buffers pointed by dest and src overlap, then the behavior of memcpy() is also undefined!

The next function, memmove() copies "n" bytes from the source buffer (src) to the destination buffer (dest). This function brings the good news that we have been waiting for -- for the case when the buffers pointed by source and destination overlaps dest and src overlap, memmove() correctly copies the source buffer to the destination buffer!

All of the above four functions return a pointer to the updated dest string.

If we were to attempt to copy a string to a string that has smaller buffer, then the behavior is undefined; in simpler words, doing so would be plain wrong! Hence, when copying strings, it is wise to pay attention to two things; the destination string should have enough buffer and the NUL character should be copied (added) at the end of the string.

Armed with a basic understanding of the above functions, let us now look at four examples to increase our understanding further. The first two examples focus on string-related functions. The last two examples focus on memory-related functions.

Examples: strcpy() and strncpy()

Our first example shows a simple usage of strcpy and strncpy functions. The example uses the built-in function strlen() to get the total number of characters present in a string. However, strlen() does not include the terminating NUL character. Accordingly, we pass an additional byte, as "len +1" to the strncpy() call, so that it also copies the terminating NUL character. Here is the example:

```
#include <stdio.h>
#include <string.h>

#define STR_LONG "Starry Nights by Vincent Van Gogh"
#define STR_SHORT "The Yellow House"

#define LEN_STRING 50

int main () {
    char var_str[LEN_STRING];
    char* str_temp;
```

```
     size_t len;

     /* Copy a Short string to a short string */
     printf("[strcpy] Copying shorter string:\n");
     printf("[strcpy] Before copy: %-35s (len: %2d)\n", var_str,
strlen(var_str));
     str_temp = strcpy(var_str, STR_SHORT);
     printf("[strcpy] After copy : %-35s (len: %2d)\n", var_str,
strlen(var_str));

     /* Copy a long string to a long string using strncpy */
     len = (strlen(STR_LONG) > LEN_STRING) ?  strlen(STR_LONG) : LEN_STRING;
     printf("\n[strncpy] Copying longer string (%d bytes):\n", len);
     printf("[strncpy] Before copy: %-35s (len: %2d)\n", var_str,
strlen(var_str));
     str_temp = strncpy(var_str, STR_LONG, (len + 1)); /* Add 1 for NUL
character */
     printf("[strncpy] After copy : %-35s (len: %2d)\n", var_str,
strlen(var_str));
     return 0;
 }
```

Note that the above program show two different ways in which we can initialize a string. First, the string, STRING_LONG, is merely a macro and therefore, where ever needed the entire string represented by STRING_LONG gets replaced by the compiler. The second style uses a string variable as an array of chars (var_str). This method provides us with the pointer to the beginning of these strings and we can navigate them from there. There is yet another way of using strings and that is by doing a malloc of a character array.

Let us now see the output of the above program to understand various parts better.

```
 $ gcc -g strcpy.c -o strcp
 $
 $ ./strcp
 [strcpy] Copying shorter string:
 [strcpy] Before copy:                                    (len: 14)
 [strcpy] After copy : The Yellow House                   (len: 16)

 [strncpy] Copying longer string (50 bytes):
 [strncpy] Before copy: The Yellow House                  (len: 16)
 [strncpy] After copy : Starry Nights by Vincent Van Gogh (len: 33)
```

As expected, both strcpy() and strncpy() copy the source string into the destination string. strcpy() automatically appends the NUL character at the end. The output shows that before we call the strcpy(), var_str variable contains garbled text (not shown) -- this is because it is not initialized. Hence, it is a good idea to initialize a string, when applicable. We can initialize strings using memset() call that we will visit in a later section.

Our second example reveals a subtle folly of the strcpy()/strncpy() functions. These functions do not work correctly if the two strings (source string and destination string) have an overlap. This example intentionally tries to do a strncpy() to a destination string, when the destination string overlaps with the source string.

The example has two strings: src (equal to "0123456789") and dest. Next, the example makes dest point to src and then advances it by 4 characters. This way, dest becomes "456789". When we copy dest worth of bytes from src to dest, due to overlap, src ends up writing to itself! Here is the example:

```
#include <stdio.h>
#include <string.h>

int main () {
    char src[] = "0123456789";
    char *dest, *str_temp;

    dest = src;
    dest += 4; /* Move the pointer ahead by 4 bytes */

    printf("Before copy:  src: %-10s (len: %3d)\n", src, strlen(src));
    printf("Before copy: dest: %-10s (len: %3d)\n\n", dest, strlen(dest));

    str_temp = strncpy(dest, src, strlen(dest));

    printf("After copy :  src: %-10s (len: %3d)\n", src, strlen(src));
    printf("After copy : dest: %-10s (len: %3d)\n", dest, strlen(dest));

    return 0;
}
```

When we run the program, the output is not the same as expected. We would have expected the output to show dest as "012345" -- instead, it shows dest as "012301"!

```
Before copy:  src: 0123456789 (len:  10)
Before copy: dest: 456789     (len:   6)

After copy :  src: 0123012301 (len:  10)
After copy : dest: 012301     (len:   6)
```

So, what went wrong? Basically, when we try to copy 6 bytes of string from src to dest, the strncpy does not account for overlap and unknowingly overwrites the storage space (shared by both variables). As noted earlier, strcpy() and memcpy() also suffer from this flaw.

Let us use a figure to explain this oddity. For the sake of clarity, the figure shows strncpy() copying in two steps. In the first step, it shows copying of the first 4 bytes to show the initial overwrite (we choose 4 bytes because dest is ahead of src by 4 bytes). In the second step, we copy the remaining 2 bytes.

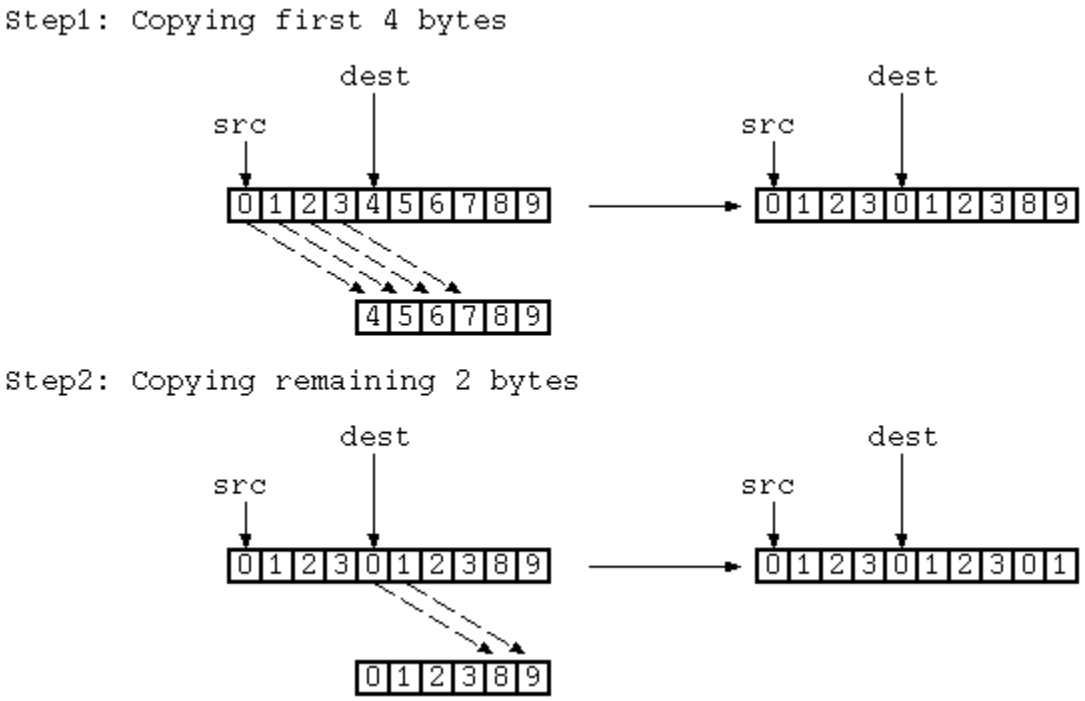


Figure: Overlapping bytes: copying 6 bytes from src to dest

As shown in the above figure, by the time the first 4 bytes are written, src ends up overwriting itself. Due to that, the value of character '4' gets overwritten by character '0', and so on. At the start of the second step, src is now "0123012389" and dest is now "012389". The second step does no good either and copies the 5th and 6th bytes ("01") back to itself leading to the final value of "0123012301"!

In case you have started to worry, there are two ways out of this problem. First and simple way is to use memmove(). Second way is to take into account the overlap and copy bytes in the correct order.

The copy implementation should consider using the correct copying order. In the earlier figure we started copying with the first character and the order was: '0' first, followed by '1', '3', '4', and '5'; this ordering leads to overwriting. If we were to copy in the reverse direction ('4' first, followed by '3', '2', '1', and '0'), we can arrive at the correct output. We show this in the figure below.

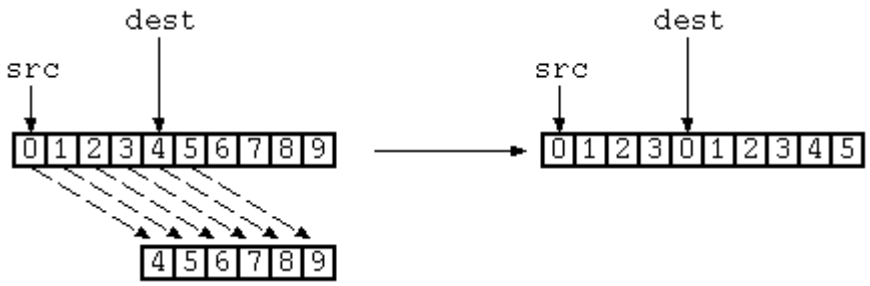


Figure: Overlapping bytes: coping in reverse order provides correct copy

The ordering should be determined based on the starting address of the two strings. If there is an overlap and the starting character of the destination string lies after the starting character of the source string, then the reverse ordering is the right approach. On the other hand, if the starting character of the destination string lies before the starting character of the source string, then the normal ordering is the right approach.

Thus, if we suspect that there might be an overlap between the source and the destination string, then we should avoid using strcpy()/strncpy(). Note that even though the behavior is undefined in this case, this does not mean that these two functions may not provide a correct behavior in some of the cases. But, if you have got luck like mine, then you are likely to see the incorrect behavior!

Examples: memcpy() and memmove()

Our last two examples go beyond the two string-related functions. They provides implementations for the two memory-related functions: memcpy() and memmove().

The following example shows usage of memcpy() and memmove(). It also demonstrates that memmove() is the right function to use when there is an overlap between the source string and the destination string.

```
#include <stdio.h>
#include <string.h>

#define STR_MONALISA "Mona Lisa was painted in the year 1505."
#define LEN_STRING 50
#define STRING_NUM "0123456789"

int main () {
    char src[LEN_STRING];
    char *str_temp, *dest;

    memcpy((void *)src, STR_MONALISA, strlen(STR_MONALISA) + 1);
    printf("[memcpy] src: %s \n\n", src);

    memcpy((void *)src, STRING_NUM, strlen(STRING_NUM) + 1);
    dest = src;
    dest += 4; /* Move the pointer a little bit ahead */
    printf("[memcpy] Before copy: src: %s, dest: %s\n", src, dest);
    str_temp = (char *)memcpy((void *)dest, (void *)src, strlen(dest));
```

```
    printf("[memcpy] After copy : src: %s, dest: %s\n\n", src, dest);

    /* Reset src to point to STRING_NUM again */
    memcpy((void *)src, STRING_NUM, strlen(STRING_NUM) + 1);
    dest = src;
    dest += 4; /* Move the pointer a little bit ahead */

    printf("[memmove] Before copy: src: %s, dest: %s\n", src, dest);
    str_temp = memmove(dest, src, strlen(dest));
    printf("[memmove] After copy : src: %s, dest: %s\n", src, dest);

    return 0;
}
```

The example starts with a string variable, str and uses memcpy() to copy string MONA_LISA_YEAR to str. Next, it uses STRING_NUM to demonstrate the case of overlap. Here is the output:

```
$ gcc memcpy.c -o memcpy
$
$ ./memcpy
[memcpy] src: Mona Lisa was painted in the year 1505.

[memcpy] Before copy: src: 0123456789, dest: 456789
[memcpy] After copy : src: 0123012345, dest: 012345

[memmove] Before copy: src: 0123456789, dest: 456789
[memmove] After copy : src: 0123012345, dest: 012345
```

The above output shows that for the case of overlapping strings, surprisingly, memcpy() also provides a correct output. As we noted earlier, when we copy "0123456789" to "456789", the output should be "0123012345". Even if the behavior is undefined, it does not mean that memcpy() would always provide an incorrect output to every copying of overlapping strings -- in some cases, it might still provide the correct output. I guess, I got lucky! However, its behavior should still be treated as undefined. When we use the memmove() variant, we find that the output is correct.

The last example shows the unique advantage that memcpy() and memmove() enjoy over strcpy() and strncpy() -- they can copy data for non-string storage types as well. The example (provided below) defines a data structure, painting_frame and use memmove() to copy the values of the information stored in one data structure to another. Since the return value of memmove() is same as the destination string, we ignore the output using the "(void)" syntax.

```
#include <stdio.h>
#include <string.h>

#define BUFFER_SIZE 100
#define STR_PAINTER "Leonardo da Vinci"

typedef struct painting_frame {
    int painting_id;
    int width;
```

```
    int height;
    char painter[BUFFER_SIZE];
} painting_frame_t;

void print_painting (painting_frame_t* p_frame) {
    if (!p_frame) return;

    printf("Printing Painting Frame Info: \n");
    printf("\tID     : %5d\n", p_frame->painting_id);
    printf("\twidth  : %5d\n", p_frame->width);
    printf("\theight : %5d\n", p_frame->height);
    printf("\tpainter: %s\n\n", p_frame->painter);
}

int main () {
    painting_frame_t painting1 = {1001, 40, 100, STR_PAINTER};
    painting_frame_t painting2;

    print_painting(&painting1);

    (void) memmove((void *)&painting2, (void *)&painting1,
sizeof(painting_frame_t));

    painting2.painting_id = 1002;
    print_painting(&painting2);

    return 0;
}
```

Here is the output of the above code.

```
$ gcc memstructcpy.c -o memstruct
$
$ ./memstruct
Printing Painting Frame Info:
       ID     :  1001
       width  :    40
       height :   100
       painter: Leonardo da Vinci

Printing Painting Frame Info:
       ID     :  1002
       width  :    40
       height :   100
       painter: Leonardo da Vinci
```

C String Library: Comparison Functions

C provides a set of library functions for manipulating both string and non-string data; these functions are published using the "string.h" header file. Where as string-related functions data use a pointer to char ("char *"), non-string-related functions use a pointer to void ("void *"). Using

Please recall that C represents strings as an array of char types, where each char type requires one byte of storage. Also, C places '\0' (NUL termination character) as the last character to mark the end of the string.

In this section, we focus on functions that compare two (string or non-string) data with each other.

Overview of Functions

C string library provides three comparison functions: two for strings and one for non-string data. Given two strings (or non-string data), these functions compare the value present present in corresponding bytes; if they are same for both, then the two strings (or non-string) data are deemed equal.

Thus, if there are two strings: str1 (with value "Mona Lisa") and str2 (with value "Mona Lisa"), then the value present in corresponding bytes is same for both -- first byte holds 'M' for both, second byte holds 'o' for both, and so on. Thus, they are equal. On the other hand, if we were to compare two strings: str1 (with value "Mona Lisa") and str3 (with value "Mona Simpson"), then these strings store different values starting with the sixth byte and hence, are unequal.

With that, let us get the ball rolling and provide signatures of the comparison functions:

```
int strcmp(const char *dest, const char *src);
int strncmp(const char *dest, const char *src, size_t n);
int memcmp(const void* dest, const void *src, size_t n);
```

The first function, strcmp(), returns 0, greater than 0, or less than 0 if the string dest is same, or greater than, or less than string src. The next function, strncmp(), is similar to that of strcmp(), but it compares only the first n bytes of string src with the first n bytes of string dest. If n is greater than the size of src, then comparison is made till the NUL-termination of the src string.

Like strncmp(), memcmp() compares the first n bytes of the src buffer with the first n bytes of dest buffer; it returns 0, greater than 0, or less than 0 to if the buffer pointed by dest is equal, greater than, or less than that of object src. Note that memcmp() already takes n size_t as input, and as such, there is no function like memncmp()!

For the case when two values are different, the sign of the nonzero value returned is determined by the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the values being compared. Thus, if we are comparing "Mona Lisa" and "Mona Simpson", then the return value would be based on the difference between 'L' and 'S'.

Examples

Let us now see three examples that provide usage of these functions. The first example uses strcmp()/strncmp() pair to compare two strings. The second example repeats the first example, but uses memcmp() instead of strcmp() and strncmp(). The third example uses memcmp() to compare non-string values.

The first example (provided below) compares a string (variable str) to two other strings (STR_LAST_SUPPER and STR_LAST_SHOW). The variable str is different than STR_LAST_SHOW and hence, strcmp() fails. However, str and STR_LAST_SHOW do have a common substring ("The Last S") and so when we use strncmp() to compare just the common substring, the comparison succeeds.

```
#include <stdio.h>
#include <string.h>

#define STR_LAST_SUPPER "The Last Supper"
#define STR_LAST_SHOW   "The Last Show"

int main () {
    char str[] = "The Last Supper";
    int ret;
    size_t len;

    ret = strcmp(str, STR_LAST_SUPPER);
    printf("[strcmp]  Returned value: %2d\n", ret);

    ret = strcmp(str, STR_LAST_SHOW);
    printf("[strcmp]  Returned value: %2d\n", ret);

    ret = strcmp(STR_LAST_SHOW, str);
    printf("[strcmp]  Returned value: %2d\n", ret);

    len = strlen("The Last S");
    ret = strncmp(str, STR_LAST_SHOW, len);
    printf("[strncmp] Returned value: %2d\n", ret);
    return 0;
}
```

We provide the output below. Note that when we change the order of strings passed to strcmp() in the second and third calls, the return value also changes.

```
$ gcc strcmp.c -o strcmp
$
$ ./strcmp
[strcmp]  Returned value:  0
[strcmp]  Returned value:  1
[strcmp]  Returned value: -1
[strncmp] Returned value:  0
```

Our next example repeats the logic of the earlier example -- the only difference is that, it uses memcmp() instead of strcpy()/strncpy(). Accordingly, we cast strings to void pointers (void *). The output is similar to that of earlier example and hence, we omit it.

```
#include <stdio.h>
#include <string.h>

#define STR_LAST_SUPPER "The Last Supper"
#define STR_LAST_SHOW   "The Last Show"
```

```
 int main () {
     char str[] = "The Last Supper";
     int ret;
     size_t lenSubstring;

     ret = memcmp((void *)str, (void *)STR_LAST_SUPPER,
strlen(STR_LAST_SUPPER));
     printf("[memcmp] Returned value: %2d\n", ret);

     ret = memcmp((void *)str, (void *)STR_LAST_SHOW, strlen(STR_LAST_SHOW));
     printf("[memcmp] Returned value: %2d\n", ret);

     ret = memcmp((void *)STR_LAST_SHOW, (void *)str, strlen(STR_LAST_SHOW));
     printf("[memcmp] Returned value: %2d\n", ret);

     lenSubstring = strlen("The Last S");
     ret = memcmp((void *)str, (void *)STR_LAST_SHOW, lenSubstring);
     printf("[memcmp] Returned value: %2d\n", ret);

     return 0;
 }
 $ gcc memcmp.c -o memcmp
 $
 $ ./memcmp
 [memcmp] Returned value:  0
 [memcmp] Returned value:  1
 [memcmp] Returned value: -1
 [memcmp] Returned value:  0
```

Compared to strcmp() and strncmp(), memcmp() does have a neat trick up its sleeve! It can also compare non-string data. In that regard, our last example compares two data structures. So, here it goes!

```
 #include <stdio.h>
 #include <string.h>

 typedef struct painting_frame {
     int painting_id;
     int width;
     int height;
 } painting_frame_t;

 int main () {
     painting_frame_t painting1 = {1001, 40, 100};
     painting_frame_t painting2 = {1002, 40, 100};
     int ret;

     /* The painting_id values are different, so comparison should fail */
     ret = memcmp((void *)&painting1, (void *)&painting2,
sizeof(painting_frame_t));
     printf("[memcmp] Returned value: %2d\n", ret);

     /* Make the painting_id values same for both and then compare again */
     painting2.painting_id = painting1.painting_id;
     ret = memcmp((void *)&painting1, (void *)&painting2,
sizeof(painting_frame_t));
```

```
    printf("[memcmp] Returned value: %2d\n", ret);
    return 0;
}
```

As expected, the comparison fails initially since the value of painting_id is different for the two data structures. For the second memcmp() call, we keep the values of the painting_id fields same and so the comparison succeeds.

```
$ gcc structmemcmp.c -o structmemcmp
$
$ ./structmemcmp
[memcmp] Returned value: -1
[memcmp] Returned value:  0
```

C String Library: Search Functions

C String library provides various functions for handling string and non-string data. These functions are published using the "string.h" header file. Where as, the string-related functions deal with char pointers (char *), the memory-related functions deal with void pointers (void *).

Let us recall that C represents strings as an array of char types -- each char type requires one byte of storage. When C stores a string in an array, it uses '\0' (NUL termination character) as the last character to mark the end of the string.

In this section, we take a look at functions that help us search for a specific value (character, string, or data) within another value (string or data).

Overview of Functions

As always, we begin with a short overview of search-related functions. The first three functions search string data and the remaining two search non-string data. However, we can use the memory-related functions even on strings by casting strings to void pointers.

```
char *strchr(const char* dest, int c);
char *strrchr(const char* dest, int c);
char *strstr(const char *dest, const char *substr);
void *memchr(const void *dest, int c, size_t n);
void *memrchr(const void *dest, int c, size_t n);
```

The function strchr() returns the first occurrence of a given character ('c') in the string dest. strrchr() returns the first occurrence of a given character, but the search begins from the reverse side of the string dest. The next function, strstr() locates the first occurrence of a substring (substr) within the dest string.

Functions memchr() and memrchr() are in someways counterparts of strchr() and strrchr() with one difference -- they can search non-string data as well. memchr() finds the first occurrence of character c in the n bytes of dest buffer; memrchr() is similar to memchr() but it searches from end of the buffer.

Even though, the above functions take "c" as an integer, the value of "c" is actually read as an unsigned char. Thus, even though an integer has multiple bytes (4 bytes in most of the platforms), "int c" is actually reads only as a single byte, since an unsigned char requires only one byte. Lastly, since bytes are stored as ASCII values and ASCII values are different for upper-case and lower-case values, these functions are case-sensitive.

If the lookup fails, then all of the above functions return NULL.

Before we go any further, we would like to mention that GNU Lib C (glibc) provide a memory equivalent function for strstr(): memmem(). memmem() finds occurrences of a given data subset within a bigger data. Since memmem() may not be available on all versions of C-releases, we do not cover it here. Nonetheless, if you are using glibc and code-portability beyond gcc is not an issue for you (at least, in the short-term!), then "memmove" may be worth exploring.

Examples

Let us now go through two simple examples and enhance our understanding of these search functions.

The first example demonstrates how we can search for a given character or a given substring, within another string. It uses all of the above functions: strchr(), strrchr(), strstr(), memchr(), and memrchr().

Each of these functions return a pointer to the location of the first occurrence of the character (or the substring) being searched; for the reverse-based functions, the first location is counted from the end. The program prints the returned value of these functions.

```
#include <stdio.h>
#include <string.h>

int main () {
    char str[] = "The Last Supper by Leonardo da Vinci";
    char *str_temp;

    str_temp = strchr(str, 'L'); /* Find a character */
    printf("[strchr]  Returned string: %s\n", str_temp);

    str_temp = strrchr(str, 'L'); /* Start search from the end */
    printf("[strrchr] Returned string: %s\n", str_temp);

    str_temp = strstr(str, "Supper"); /* Find a substring in the string */
    printf("[strstr]  Returned string: %s\n\n", str_temp);

    /* Following calls use memory APIs for the above tasks */
    str_temp = (char *)memchr((void *)str, 'L', strlen(str));
    printf("[memchr]  Returned string: %s\n", str_temp);

    str_temp = (char *)memrchr((void *)str, 'L', strlen(str));
    printf("[memrchr] Returned string: %s\n", str_temp);
    return 0;
}
```

Note that, if we were to search for an non-existing character (e.g. 'Z'), or a non-existing substring (e.g. "Mona"), then these functions would return a NULL. Accordingly, the calling function should check if the returned value is NULL and process the returned value only when it is not NULL. Here is the output.

```
$ gcc strsearch.c -o strsearch
$
$ ./strsearch
[strchr]  Returned string: Last Supper by Leonardo da Vinci
[strrchr] Returned string: Leonardo da Vinci
[strstr]  Returned string: Supper by Leonardo da Vinci

[memchr]  Returned string: Last Supper by Leonardo da Vinci
[memrchr] Returned string: Leonardo da Vinci
```

Our discussion for search-related functions would be incomplete if we do not see an example of how memory search works for non-string data! With that goal in mind, let us write our second example uses memchr() to search a character within a data structure.

The example begins with a definition of a simple data structure and then uses memchr() to see if a character is present in the data structure or not. Since the returned value does not point a string, we cannot use printf() to print the returned value. Bounded by this constraint, we choose a workaround and instead, print the address of the data structure and the returned value.

```
#include <stdio.h>
#include <string.h>

#define BUFFER_SIZE 100
#define STR_PAINTER "Leonardo da Vinci"

typedef struct painting_frame {
    int painting_id;
    int width;
    int height;
    char painter[BUFFER_SIZE];
} painting_frame_t;

int main () {
    painting_frame_t painting = {1001, 40, 100, STR_PAINTER};
    void *void_temp;

    void_temp = memchr((void *)&painting, 'L', sizeof(painting_frame_t));
    printf("Address of data-structure: %p\n", &painting);
    printf("Address of 'L' character : %p\n", &painting.painter);
    printf("Returned location for 'L': %p\n", void_temp);

    void_temp = memchr((void *)&painting, 'a', sizeof(painting_frame_t));
    printf("Returned location for 'a': %p\n", void_temp);

    return 0;
}
```

When we run this program, we find that memchr() returns a pointer to the location of 'L' -- this location is located at an offset of 12 bytes from the address of the painting data structure. This is because, 'L' is at the start of the painter field of the structure and prior to that, the structure has three integer fields. On the system, where this is compiled, the size of an integer is 4 bytes and so the offset is 12 bytes. If we run this on a system, where an integer requires 8 bytes, then the offset would be 24 bytes. The pointer to 'a' character is 4 bytes further down that of 'L'. Here is the output.

```
Address of data-structure: 0xbf8ce0ec
Address of 'L' character : 0xbf8ce0f8
Returned location for 'L': 0xbf8ce0f8
Returned location for 'a': 0xbf8ce0fc
```

Before we conclude, we should note that the painting_frame structure is well-aligned -- each of the int fields are of sizes 4 bytes and follow the alignment boundary. Even the painter array has a size of 100, which can also be aligned with 4 bytes. However, if this were not the case and if painting_frame was not aligned (e.g. by adding a "char val[3]" at the start of the structure), then the compiler would add some padding bytes. In that cases, the above offsets would also need to take into account compiler-added alignment padding.

This example is provided purely for the sake of helping us understand memchr() for non-string data. Before accessing data structure members using raw memory, we should pay close attention to alignment for data structure. For more on data structure alignment, please visit our page on C data structures.

C String Library: Miscellaneous Functions

C provides a set of library functions for manipulating string and non-string data. These functions use a char pointer ("char *") for string data and a void pointer ("void *") for non-string data. C publishes them via the "string.h" header file. We should include this header file whenever we use these functions.

Please recall that a C string is an array of char types, where each char type requires one byte of storage. Equally important, when C stores a string in an array, it uses '\0' (NUL termination character) as the last character to mark the end of the string.

This section describes some of the miscellaneous functions provided by string library like concatenating strings, tokenising strings, etc.

Overview of Functions

As usual, we begin with the functions prototype for these miscellaneous string-related APIs.

```
size_t  strlen(const char *s);
char    *strcat(char *dest, const char *src);
char    *strncat(char *dest, const char *src, size_t n);
char    *strtok(char *dest, const char *delimiter);
```

```
char    *strerror(int errnum);
void    *memset(void *dest, int c, size_t n);
```

The first function is strlen() and we have already seen it working in earlier sections! Basically, this function returns the total number of characters (that is, the length) of a string, excluding the last NUL-termination byte.

The next two functions strcat() and strncat(), append destination string (dest) to the source string (src). Where as, strcat() appends entire string src to the end of string dest, strncat() does so only for the first n characters of string src. If the length of the string src is less than n, then it concatenates the entire string.

For both cases, dest string should have enough space to accommodate both the strings and the NUL character. Hence, for strcat(), the dest string should have at least (strlen(dest) + str(src) + 1) bytes of space. For strncat(), it should have at least (strlen(dest) + n + 1) bytes of space; if n is greater than the length of src, then the dest string should have at least (strlen(dest) + strlen(src) + 1) bytes of space.

Both of these functions remove the NUL character from the end of dest string before appending the src string and then, once again add the NUL character at the end. If the size n is greater than the length of the src string, then strncat() copies entire src string and then appends the NUL character at the end. Needless to say, this is done to make the combined character array a new string because C uses the NUL character to mark the end of a string.

Both strcat and strncat, return the dest string after appending. Lastly, the src and dest strings should not overlap, otherwise the behavior is undefined.

The next function, strtok(), splits dest string into tokens that are separated by a string delimiter. We need to call strtok() multiple times to retrieve all the tokens. The function strerror() returns the string of the error name that is associated with the error number, errnum. We typically pass the system error number (errno) to this function.

Lastly, memset() copies character c into the first n byte of dest buffer. This function returns the updated dest string. We can use memset() even on strings, as long as we cast the strings to void pointers.

Examples

Having described these string-based functions, let us now delve into the fun part of their implementation! For that, we provide two examples. The first example uses strcat() and strncat() to concatenate two strings. The second example focuses on the remaining functions.

The first example (provided below) has two parts. It starts by calling strcat() to concatenate a string (STR_TO_APPEND1) to another string (str_dest1) -- str_dest1 has enough storage (100 bytes) to hold both of these strings. Next, it uses malloc to create a string buffer big enough to hold two strings and then uses strncat() to append both strings, one after another. We include the "stdlib.h" header file for the malloc() and free() calls.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define STR_TO_APPEND1 "was painted by Leonardo da Vinci"
#define STR_TO_APPEND2 "was painted by Van Gogh"

int main () {
    char str_dest1[100] = "Mona Lisa ";
    char str_dest2[] = "Starry Night ";
    char *str_cat, *str_malloc;

    /* Append to an existing string buffer */
    str_cat = strcat(str_dest1, STR_TO_APPEND1);
    printf("[strcat]  Returned string: %s\n", str_cat);
    printf("[strcat]  str_cat: %p, str_dest1: %p\n\n", str_cat, str_dest1);

    /* Malloc a string buffer and then append a string to it */
    str_malloc = (char *)malloc(sizeof(char) *
              (strlen(str_dest2) + strlen(STR_TO_APPEND2) + 1));
    if (!str_malloc) return -1;

    str_cat = strncat(str_malloc, str_dest2, strlen(str_dest2));
    str_cat = strncat(str_malloc, STR_TO_APPEND2, strlen(STR_TO_APPEND2));
    printf("[strncat] Returned string: %s\n", str_cat);
    printf("[strncat] str_cat: %p, str_malloc: %p\n", str_cat, str_malloc);

    /* Free the malloced string */
    free(str_malloc);
    return 0;
}
```

At the cost of repeating myself, for concatenation operations, the destination string must have enough space to accommodate both the strings and the NUL character; else, the behavior would be undefined.

We provide below the output. Note that when we print the address of the string returned by both of these functions, then we find that it is the same string as that of the destination string (since both of them point to the same address).

```
$ gcc strcat.c -o strcat
$
$ ./strcat
[strcat]  Returned string: Mona Lisa was painted by Leonardo da Vinci
[strcat]  str_cat: 0xbfbc9454, str_dest1: 0xbfbc9454

[strncat] Returned string: Starry Night was painted by Van Gogh
[strncat] str_cat: 0x9112008, str_malloc: 0x9112008
```

Our second and last example shows the usage of memset(), strlen(), strtok(), and strerror(). The program begins by setting characters in str_a to zero using memset. Next, it uses strtok() to split str_b into tokens. For strtok() function, we pass white-space (" ") as the delimiter and thus, strtok() returns each word of the sentence as a token. Also, it is only the first call to strtok(),

where we pass the string as input -- subsequent calls take NULL as input and retrieve the earlier string from memory.

In the end, the program passes the system (global) errno variable to strerror() function; errno contains the last error encountered by the system (this system-wide variable is defined in "errno.h" header file).

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main () {
    char str_a[] = "The Last Supper by Leonardo da Vinci";
    char str_b[] = "The Last Supper";
    char *str_temp;

    printf("[memset] Before setting to zero, str: %s\n", str_a);
    memset((void *)str_a, 0, strlen(str_a));
    printf("[memset]  After setting to zero, str: %s\n", str_a);

    /* Split the string into tokens */
    printf("\n[strtok] Let us tokenize the string: %s\n", str_b);
    str_temp = strtok(str_b, " ");
    while (str_temp != NULL) {
        printf("[strtok] Returned token: %s\n", str_temp);
        str_temp = strtok(NULL, " ");
    }

    printf("\n[strerror] The error string: %s\n", strerror(errno));
    return 0;
}
```

We provide the output below. Since, there is no error, the strerror() call returns "Success" string when we passed errno to it!

```
$ gcc strmisc.c -o strmisc
$
$ ./strmisc
[memset] Before setting to zero, str: The Last Supper by Leonardo da Vinci
[memset]  After setting to zero, str:

[strtok] Let us tokenize the string: The Last Supper
[strtok] Returned token: The
[strtok] Returned token: Last
[strtok] Returned token: Supper

[strerror] The error string: Success
```

POSIX Threads: Pthread Basics

POSIX threads (Pthreads) is a widely used standard for writing multi-threaded programs. Although, technically not a part of C, POSIX threads are commonly used for both C and C++ applications. Hence, we include discussion on Pthreads.

An operating system process can have multiple threads. Multiple threads allow the process to do tasks concurrently. Even better, if there are multiple processors, then these threads can run on different multiple processors, and thereby, reach (almost) true parallelism. Hence, an application with tasks that can be done independently, or with little (bounded) dependency on each other, can greatly benefit from using threads.

Let us consider an example of a web-server that handles multiple HTTP requests. The web-server can spawn multiple threads and each thread can handle a single request. At the same time, another thread can listen for incoming HTTP requests. Thus, threads can enable a web-server to not only listen for incoming requests, but also to process existing requests -- and all of these happen concurrently.

Threads belonging to a process share the address space, file descriptors, etc belonging to the parent process with each other. However, each thread does maintain its own stack so that it can execute common lines of code independently of other threads. Since threads do not have to maintain their own address space etc, they are also referred to as light weight processes!

One naive approach may be to run concurrent tasks as multiple processes. But, switching between different processes (aka scheduling) has a lot more overhead than switching between threads within one process. No less important, communication between different processes would require some form of inter process communication (IPC) which again has a lot more overhead, not to mention more complicated. By comparison, communication between threads is simpler since they share the address space, file descriptors belonging to the parent process.

Of course, when we have multiple threads working together, there is always a risk of them overwriting any common memory location (or critical data). Therefore, isolating the critical data in the memory and making threads access them in a synchronized manner goes hand in hand with creating threads. Pthreads offer mutex -- short for mutual exclusion -- to provide synchronization for multiple threads. Lastly, threads also need to communicate with each other. To facilitate inter-thread communication, Pthreads provide conditional variables (or condvars).

Getting started

Pthreads provide a new variable type for threads: pthread_t. Thus, if we have to define a Pthread variable, x, then we can do it as "pthread_t x". pthread_t type is simply an integer (in most of the platforms) that stores an identifier value of the thread. More importantly, pthread_t is an opaque value, so we should not make any assumption about its internals, else, the resulting code may cease to be portable.

Let us get started by providing the basic APIs that create threads and allows the parent threads to wait for them till they are done executing their tasks. Here is the signatures of these APIs:

```
int pthread_create(pthread_t *t,
                   const pthread_attr_t *a,
                   void *(*target_func)(void *),
                   void *a);
int pthread_join(pthread_t t, void **value_ptr);
```

The pthread_create() API is truly the mother of all Pthread APIs, since it is this API that brings a thread into existence! More specifically, pthread_create() creates a new (child) thread and puts the thread identifier into the first parameter, which is a pointer to a pthread_t variable. The function also takes three additional parameters. The attr parameter allows us to specify attributes for the new thread -- if we choose default attributes, then we can keep this parameter as NULL. The target_func parameter specifies a target function and once created, the newly minted thread's job will be to run this function.

The last parameter of pthread_create() takes a pointer to a value that can be passed as an argument to the target function. This will work easily, if we need to pass a single value to the target_func -- basically pass a pointer to the value (aka variable) and of course cast it as a void pointer. However, when we have more than one value to pass to the target_func, then this approach would not work. In that case, one solution is define a data structure and use that as a container to hold multiple values and then pass the (void-casted) pointer of that data structure.

Function pthread_create() returns 0 if is successful, and a non-zero error code, if not. It is a good idea to always check the return value and if error happens, then we should print the returned value for debugging purposes -- multi-threaded programs are notoriously difficult to debug and therefore, so we should go the extra mile to get all the error information logged.

While the child thread finishes its task, the caller thread can (and often should) wait for the child thread to complete its task. In most cases, completing the task means returning from the target function. POSIX provides pthread_join() API that suspends the caller thread till the child thread is done; when the child thread returns from the target function, it is put in a terminated state. If the main thread finishes its completion and does not wait for the (child) threads, then the child thread (along with other resources of the main thread) would be destroyed immediately -- so, in many cases, the polite thing would be for the caller thread to simply wait.

The second API in the above set, pthread_join() takes pthread_t identifier of the thread for which we wish to wait. It returns 0 if the thread is done with its task and the join succeeds. If not, then it returns an error. Once again, the good thing to do is to report the status, if it is not zero. One common reason why pthread_join() can fail is if the child thread ceases to exist. This error can happen, if the child thread detaches itself (we will talk about detaching threads in a moment).

Function pthread_join() also takes a second argument, value_ptr. For cases, where the child thread returns from the target function using pthread_exit(), the thread can pass a return value to pthread_exit() call and that value gets stored in the value_ptr. This is one way to provide communication between threads, albeit rather limited.

Lastly, only one thread should wait for the termination of a given child thread. If multiple threads were to call pthread_join for a given thread, then it can also lead to an error.

Let us now get our feet wet and write our first threaded program. So, here it goes:

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>

char *arrPaintings[] = {"The Last Supper", "Mona Lisa", "Potato Eaters",
                        "Cypresses", "Starry Night", "Water Lilies"};

void *selectPainting (void *arg) {
    int index = *(int *)arg;

    printf("\tPassed index is %d\n", index);
    printf("\tGoing to sleep..\n");
    sleep(10);
    printf("\tWoke up\n", arrPaintings[index]);
    printf("\tPainting is %s\n", arrPaintings[index]);
}

int main () {
    pthread_t t;
    int status;
    int arrLen = sizeof(arrPaintings)/sizeof(arrPaintings[0]);
    int index = 2;

    printf("Starting the child thread..\n");
    status = pthread_create(&t, NULL, selectPainting, (void*)&index);
    if (status != 0) {
        fprintf(stderr, "pthread_create() failed [status: %d]\n", status);
        return 0;
    }

    printf("Waiting for the child thread..\n");
    status = pthread_join(t, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_join() failed [status: %d]\n", status);
    }
    printf("Child thread is done\n");
    return 0;
}
```

Let us now understand the above program. The program begins by including <pthread.h> header file (besides <stdio.h> and <time.h>) since <pthread.h> contains definitions of Pthread APIs and the related constants. This header file also includes definitions for Pthread mutexes and conditional variables.

The main() function calls pthread_create() to create a new thread, identified by the variable t. We choose to use default attributes for the thread and pass second argument as NULL. The third argument is the target function: selectPainting() and we pass an index as the parameter to this target function. Since index is an integer variable, we pass its address after casting it as (void *). Accordingly, selectPainting casts its back to an (int *) to retrieve its value.

We have kept the logic of this program simple so as to illustrate the concepts more clearly. The program has a global array, arrPaintings that contains strings for some well-known paintings. The main() function passes an index and the target function uses that index to locate the element in the array. To demonstrate the waiting period for pthread_join(), we add a sleep() statement in the target function.

Also note that the original process runs the main() function as a thread as well. So, in the above example, we actually have two threads. And since main thread is also a thread, it can also use Pthread APIs, just like every other thread!

However, there are differences between the main thread and a child thread. First, if the main thread returns, then all the threads that are still running would be terminated and their resources would be freed. Second, the arguments passed to main are typically "void *argv" and "int argc", where as the child thread's argument is only a "void *arg".

When we compile and run the program, the output (provided below) shows that the main thread suspends as long as the child thread does not return from the target function. And when it does, the pthread_join() wait is over and the main() function returns as well. Please note that we use pthread option with gcc to add support for Pthreads library.

```
[user@codingtree]$ gcc -pthread single_thread.c -o single-thread
[user@codingtree]$
[user@codingtree]$ ./single-thread
Starting the child thread..
Waiting for the child thread..
    Passed index is 2
    Going to sleep..
    Woke up
    Painting is Potato Eaters
Child thread is done
[user@codingtree]$
```

While the above program is a good start, real-life threaded programs are rarely single-threaded! So, let us crank it up a notch and rewrite the above example to spawn multiple threads.

Our new program is same as before, but with a few differences. First, it uses an array to hold multiple Pthreads. Second, it uses another array to hold various index values. Third, it generates a random number as an index for each thread -- so that each thread can pick a random array element.

We need to hold index values in an array because, that way, we can have different values and thus different pointers, when passing them to the target function. If we were to use index as a single variable and change its value in the loop, then it is possible that different threads might refer to the same index since the pointer would be the same. The reason why this is possible is because we cannot guarantee the order of execution of different threads. It is possible that multiple threads start running the target function at the same time and so would end up using the same value of index!

The program uses a for loop for both creating new threads and for waiting till they return. In the second loop, the main thread first wait for the thread identified by thread[0], and then for the thread identified by thread[1], and so on. Thus, even if thread[1] finishes first, the main thread would continue to wait for thread[0] to finish! Needless to say, we can wait for the threads in the reverse order as well.

```
 #include <stdio.h>
 #include <pthread.h>
 #include <time.h>

 #define MAX_THREADS 2

 char *arrPaintings[] = {"The Last Supper", "Mona Lisa", "Potato Eaters",
                         "Cypresses", "Starry Night", "Water Lilies"};

 void *selectPainting (void *arg) {
     int index = *(int *)arg;

     printf("\t[Array Index: %d] Going to sleep..\n", index);
     sleep(10);
     printf("\t[Array Index: %d] Woke up. Painting: %s\n",
             index, arrPaintings[index]);
 }

 int main () {
     pthread_t t[MAX_THREADS];
     int index[MAX_THREADS];
     int status, arrLen, i;

     arrLen = sizeof(arrPaintings)/sizeof(arrPaintings[0]);

     srand(time(NULL));                  /* initialize random seed */
     for (i = 0; i < MAX_THREADS; i++) {
         index[i] = rand() % arrLen;     /* Generate a random number less
than arrLen */

         printf("[Array Index: %d] Starting the child thread..\n", index[i]);
         status = pthread_create(&t[i], NULL,
                          selectPainting, (void*)&index[i]);
         if (status != 0) {
             fprintf(stderr, "pthread_create() failed [status: %d]\n",
status);
             return 0;
         }
     }

     for (i = 0; i < MAX_THREADS; i++) {
         printf("[Array Index: %d] Waiting for the child thread..\n",
index[i]);
         status = pthread_join(t[i], NULL);
         if (status != 0) {
             fprintf(stderr, "pthread_join() failed [status: %d]\n", status);
         }
         printf("[Array Index: %d] Child thread is done\n", index[i]);
     }
```

```
}
```

We provide the output below. We see that the main thread chooses two random array indices (2 and 5). Here is the sequence of how these threads run: the first thread (working on array index 2) runs first, the main thread starts to wait, the second thread starts to run. Once done, both threads return to the main thread and it is then that the main thread return.

```
[Array Index: 2] Starting the child thread..
[Array Index: 5] Starting the child thread..
    [Array Index: 2] Going to sleep..
[Array Index: 2] Waiting for the child thread..
    [Array Index: 5] Going to sleep..
    [Array Index: 2] Woke up. Painting: Potato Eaters
    [Array Index: 5] Woke up. Painting: Water Lilies
[Array Index: 2] Child thread is done
[Array Index: 5] Waiting for the child thread..
[Array Index: 5] Child thread is done
```

An important point worth mentioning is that in the above program, even though we have multiple threads sleeping for 10 seconds each (we call the sleep() function in the target function), we would see the main thread waiting (approximately) for 10 seconds and not a multiple of 10 seconds since the threads are running concurrently. You are free to take a stopwatch and verify it!

Miscellaneous APIs

Let us now look at the next set of Pthread APIs that provide miscellaneous functions. As usual, we start with the signature of these functions:

```
int pthread_exit(void *value_ptr);
int pthread_detach(pthread_t t);
int pthread_self(void);
int pthread_equal(pthread_t t1, pthread_t t2);
int sched_yield(void);
```

The first API in the above set, pthread_exit() terminates the current thread without terminating the process. This is helpful, if the current thread has child threads that still need to do some work. Of course, the process would automatically terminate when the last running thread terminates. Note that pthread_exit() is different than an exit() call since the exit() call is more extreme and terminates the process along with all of its threads. An important caveat is that if the main thread passes a pointer (as (*void) argument) to the target function and then exits using pthread_exit(), the pointer may access a dangling data. So, for such cases, using pthread_join() to suspend the main thread is a safer bet.

It is worth mentioning that when a child thread terminates (let us say, by returning from the target function), its memory resources are not released until a thread performs pthread_join on it. More specifically, pthread_join() call transitions the child thread from terminated to detached state. It is in detached state, that the we recover all the resources held by that thread. Therefore, pthread_join is also a good style to avoid memory leaks from the spawned threads. Of course, if

the main thread terminates itself, then all of the resources are automatically recovered and that is a whole different story.

A child thread can also pass a value to pthread_exit() that gets copied to the pthread_join() parameter. This is one way for a child thread to pass a return value back to the parent thread.

The next API, pthread_detach() allows us to decouple a child thread from the thread that created it. There could be use cases, where once a thread is created, it can do independent things and the creator thread does not need to communicate with it or manage (read pthread_join()) it. Once a detached thread completes its task, its resources are automatically freed.

Since pthread_join() moves a thread to a detached state, we cannot call pthread_join() on a thread that is already detached; if we do, pthread_join() is likely to greet us with an error! One likely error is ESRCH that means pthread_join() could not find the (detached) thread. Thus, we should avoid calling pthread_join() twice on the same thread!

Yet another way to detach a thread is to set the detachable attribute of the thread to true; we can do so by passing an attribute to pthread_create(). Here is a small snippet that updates the pthread_create() call from the earlier examples to include pthread attribute, indicating that the thread is detached.

```
 pthread_t;
 int status;
 pthread_attr_t attr;

 status = pthread_attr_init(&attr);
 if (status != 0) {
     fprintf(stderr, "pthread_attr_init() failed [status: %d]\n", status);
     return 0;
 }

 status = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 if (status != 0) {
     fprintf(stderr, "pthread_attr_setdetachstate() failed [status: %d]\n",
status);
     return 0;
 }

 status = pthread_create(&t, &attr, selectPainting, (void*)&index);
 ...
 ...
 /* In the end, call pthread_attr_destroy() to destroy the attr object */
 status = pthread_attr_destroy(&attr);
 if (status != 0) {
     fprintf(stderr, "pthread_attr_destroy() failed [status: %d]\n", status);
     return 0;
 }
```

The next three functions in the above list are pthread_self(), pthread_equal(), and sched_yield().

The pthread_self() returns the thread identifier (a pthread_t variable) to the current thread. In other words, if the current thread needs to get a handle of its own, then it call pthread_self().

The next function, pthread_equal() compares two pthread identifiers (phread_t variables) and if belong to the same thread, then it returns 0.

The last function, sched_yield() allows us to yield the processor to another thread/process. This provides other threads/processes a chance to run when (heavily) contended resources (e.g., mutexes) have been released by the caller. For example, if the current thread takes too long to finish its task, then other threads/processes can potentially starve. Needless to say, the threshold of deciding when it is too long is application specific. Note that if the current thread is the only thread running, then sched_yield() would continue to run the current thread since there is nobody else to yield to!

If we look at all the Pthread APIs described on this page, then we might notice that all of the APIs accept pthread_t variables by value; the only exception is pthread_create(), where we pass a pointer to a pthread_t variable. The reason is that pthread_create() needs the thread identifier of the new thread and one way to do that is by making the caller thread pass a pointer to the pthread_t variable.

Now, that we have understood the above APIs, let us look at two examples that help us understand the behavior pthread_exit().

The first example demonstrate that if we call pthread_exit() from the main thread, then the child thread continues to run even though the main thread does not call pthread_join(). This is because pthread_exit() terminates the thread (in this case, the main thread), but does not terminate the process. If we were to remove the pthread_exit() call in the following example, then the child thread would terminate immediately.

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>

char *arrPaintings[] = {"The Last Supper", "Mona Lisa", "Potato Eaters",
                        "Cypresses", "Starry Night", "Water Lilies"};

void *selectPainting (void *arg) {
    int index = *(int *)arg;

    printf("\tPassed index is %d\n", index);
    printf("\tGoing to sleep..\n");
    sleep(10);
    printf("\tWoke up\n", arrPaintings[index]);
    printf("\tPainting is %s\n", arrPaintings[index]);
}

int main () {
    pthread_t t;
    int status;
    int arrLen = sizeof(arrPaintings)/sizeof(arrPaintings[0]);
```

```
    int index = 2;

    printf("Starting the child thread..\n");
    status = pthread_create(&t, NULL, selectPainting, (void*)&index);
    if (status != 0) {
        fprintf(stderr, "pthread_create() failed [status: %d]\n", status);
        return 0;
    }

    printf("Calling pthread_exit() from the main thread\n");
    pthread_exit(NULL);
    printf("This should not be printed\n");
    return 0;
}
```

The output (provided below) confirms that even though the main thread calls pthread_exit() and returns, the child thread continues to run! In fact, it would not matter if we make the child thread detached or not since in either case, the child thread would run to completion. Note that the printf statement after pthread_exit() does not run since the main thread exits from the pthread_exit() call itself.

```
Starting the child thread..
Calling pthread_exit() from the main thread
    Passed index is 2
    Going to sleep..
    Woke up
    Painting is Potato Eaters
```

In the second example, we show communication between the child thread and the main thread using the pthread_exit() and pthread_join() combination. This time, we call pthread_exit() from the child thread and pass a pointer to a global variable. To "catch" the return value, we use a pthread_join() call in the main thread. Once the child thread returns, the pthread_join() call from the main thread unblocks and accesses the global data.

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <string.h>

char *arrPaintings[] = {"The Last Supper", "Mona Lisa", "Potato Eaters",
                        "Cypresses", "Starry Night", "Water Lilies"};

char globalStr[100];

void *selectPainting (void *arg) {
    int index = *(int *)arg;

    printf("\tPassed index is %d\n", index);
    printf("\tGoing to sleep..\n");
    sleep(10);
    printf("\tWoke up\n", arrPaintings[index]);
    printf("\tPainting is %s\n", arrPaintings[index]);

    memmove(globalStr, arrPaintings[index], strlen(arrPaintings[index]));
```

```
    pthread_exit(&globalStr);
}

int main () {
    pthread_t t;
    int status;
    int arrLen = sizeof(arrPaintings)/sizeof(arrPaintings[0]);
    int index = 2;
    char *str;

    printf("Starting the child thread..\n");
    status = pthread_create(&t, NULL, selectPainting, (void*)&index);
    if (status != 0) {
        fprintf(stderr, "pthread_create() failed [status: %d]\n", status);
        return 0;
    }

    printf("Waiting for the child thread..\n");
    status = pthread_join(t, (void **)&str);
    if (status != 0) {
        fprintf(stderr, "pthread_join() failed [status: %d]\n", status);
    }
    printf("Child thread is done\n", str);
    printf("Returned value from the Child thread: %s \n", str);
    return 0;
}
```

While the above example shows the case of the thread returning a value to the main thread using pthread_exit() and pthread_join(), we should emphasize that this approach is a trivial and lightweight form of communication among threads. For many applications, we are more likely to use a global data synchronized by mutex.

Here is the output:

```
Starting the child thread..
Waiting for the child thread..
    Passed index is 2
    Going to sleep..
    Woke up
    Painting is Potato Eaters
Child thread is done
Returned value from the Child thread: Potato Eaters
```

States of a Thread

Before we move on, let us talk briefly about various states in the life of a thread. A thread spends its entire lifetime in (mainly) four states: ready, running, blocked, and terminated.

When a thread is created using the pthread_create(), it starts its life-cycle in the ready state and would start running as and when the processor becomes available. Once the processor gives the green light, the thread runs the target function provided by the pthread_create() call.

When running, a thread can move to a blocked state, it it needs to wait for a resource (e.g. trying to acquire a mutex or waiting for an I/O that is not available immediately). Once the resource becomes available, the thread once again moves to the ready state -- this means, it is all set to run as soon as the processor becomes available.

When a thread returns from its target job or has been canceled, then it sits in the terminated state. A thread can return from the target function, either by completing the function or by pthread_exit().

In the terminated state, a thread still holds the resources that were allocated to it. If we detach the thread using pthread_join(), then its resources are recovered. Alternatively, if the thread was already detached before finishing its job or before being canceled, then its resources are recovered immediately after it reaches the terminated state. Thus, we should make sure to call pthread_join() on a thread that has not been detached.

POSIX Threads: Mutexes

With multiple threads, it is possible that one or more of them may end up accessing a common data (aka critical data) in the memory. If that happens, then we need to synchronize threads. Informally, synchronization means deciding which thread gets to access critical data first. All other threads simply need to play the waiting game; more specifically, these threads sit in the blocked state.

Pthreads achieve synchronization using mutex (short for mutual exclusion). Pthreads provides a new variable type for mutexes: pthread_mutex_t. Thus, if we have to define a Pthread mutex variable, x, then we can do it as "pthread_mutex_t x".

The easiest way to understand a mutex is to understand it as a lock such that only one thread can lock (or acquire) it at a given time. The thread that acquires the lock, gets to access the common data. In the meantime, all other threads simply wait for their turn. Once the thread with the lock is done accessing data, it can unlock the mutex. With that, one of the waiting threads will acquire the lock and access the data. And, thus the cycle of wait for mutex, lock mutex, access critical data, and unlock mutex continues. It is common to use the terms "lock a mutex" and "acquire a mutex" interchangeably.

First off, let us provide signatures of some of the common mutex APIs.

```
int pthread_mutex_lock(pthread_mutex_t *m);
int pthread_mutex_unlock(pthread_mutex_t *m);
int pthread_mutex_trylock(pthread_mutex_t *m);
int pthread_mutex_init(pthread_mutex_t *m, pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *m);
```

The first function in the above set, pthread_mutex_lock() locks a mutex. If the mutex is already locked, then the thread will wait for the mutex to becomes unlock. More specifically, the thread moves from running state to a blocked state and continues to remain blocked until the mutex is unlocked.

The next function, pthread_mutex_unlock() allows a thread to unlock a previously acquired mutex. Mutex is thread-specific and so, it can be unlocked only by the thread that holds the mutex; this thread is also referred to as the owner thread. The bad news is that if the owner thread were to terminate without unlocking the mutex, then the mutex would live in a locked state, sadly ever after! So, it is worth paying attention to threads when they terminate or exit and check if they are holding any mutex.

Since pthread_mutex_lock() would potentially block the current thread, sometimes a thread can use a handy shortcut to check if the mutex is locked or not. The thread can use pthread_mutex_trylock() to lock the mutex if it is available, else, this function returns immediately with a status of EBUSY. In the case of mutex being locked, the thread can go on and do something else and can retry later.

The last two APIs allow us to initialize and destroy a mutex variable dynamically. We can use pthread_mutex_init() to initialize a mutex dynamically. Once done, we should call pthread_mutex_destroy() to release resources attached with the mutex.

It is also possible for us to define (and initialize) a mutex variable statically (meaning that the scope is only the current file) using the PTHREAD_MUTEX_INITIALIZER macro; this macro contains a default values for various attributes of a mutex. Thus, if want to define a mutex statically, then we can do that as: "pthread_mutex_t x = PTHREAD_MUTEX_INITIALIZER;". For statically defined mutexes, there is no need to call pthread_mutex_destroy().

Before we go any further, let us see an example that uses Pthread mutex to synchronize two threads. The example mimics a paintings gallery, where paintings by various artists are continuously bought and then sold to art connoisseurs. Here it is:

```
 #include <stdio.h>
 #include <pthread.h>
 #include <time.h>
 #include <stdbool.h>

 #define TOTAL_TRANSACTIONS 5
 #define MAX_SLEEP_SECONDS_PAINTINGS_IN 10
 #define MAX_SLEEP_SECONDS_PAINTINGS_OUT 5

 static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

 static int totalPaintings = 0;
 static bool allTransactionsDone = false;

 /* Sell the paintings from the inventory */
 void *paintingsOut (void *arg) {
     int random_time;

     srand(time(NULL));
     while ((allTransactionsDone == false) || totalPaintings) {
         random_time = rand() % MAX_SLEEP_SECONDS_PAINTINGS_OUT;
         printf("\t\t[%s]Sleep for %d seconds.\n", __FUNCTION__,
random_time);
         sleep(random_time);
```

```
        pthread_mutex_lock(&mutex);
        if (!totalPaintings) {
            printf("\t\t[%s]No more paintings left. Let us retry.\n",
__FUNCTION__);
            pthread_mutex_unlock(&mutex);
            continue;
        }

        totalPaintings--;
        printf("\t\t[%s]Sold one painting. Total paintings left: %d\n",
                __FUNCTION__, totalPaintings);
        pthread_mutex_unlock(&mutex);
    }
}

/* Buy painting from artists and add to the Inventory */
void *paintingsIn (void *arg) {
    int i, random_time;

    srand(time(NULL));
    for (i = 0; i < TOTAL_TRANSACTIONS; i++) {
        random_time = rand() % MAX_SLEEP_SECONDS_PAINTINGS_IN;
        printf("\t[%s]Sleep for %d seconds. \n", __FUNCTION__, random_time);
        sleep(random_time);

        pthread_mutex_lock(&mutex);
        totalPaintings++;
        printf("\t[%s]Added one painting. Total paintings are: %d\n",
                __FUNCTION__, totalPaintings);
        pthread_mutex_unlock(&mutex);
    }
    allTransactionsDone = true;
}

int main () {
    pthread_t threadPaintingsIn, threadPaintingsOut;
    int status;

    status = pthread_create(&threadPaintingsIn, NULL, paintingsIn, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_create() failed [status: %d]\n", status);
        return 0;
    }

    status = pthread_create(&threadPaintingsOut, NULL, paintingsOut, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_create() failed [status: %d]\n", status);
        return 0;
    }

    printf("Waiting for the threadPaintingsIn..\n");
    status = pthread_join(threadPaintingsIn, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_join() failed for threadPaintingsIn
[%d]\n", status);
    }
```

```
     printf("Waiting for the threadPaintingsOut..\n");
     status = pthread_join(threadPaintingsOut, NULL);
     if (status != 0) {
         fprintf(stderr, "pthread_join() failed for threadPaintingsOut
[%d]\n", status);
     }
     return 0;
 }
```

Let us understand various pieces of the above program.

The main thread defines two threads: threadPaintingsIn and threadPaintingsOut. The thread threadPaintingsIn mimics buying of paintings from artists by incrementing a global variable, totalPaintings. The thread threadPaintingsOut mimics selling of paintings to end-users by decrementing the same global variable, totalPaintings. Both threads sleep for a while and then use a common mutex to protect this variable.

For simplicity sake, the threadPaintingsIn does not run forever. It runs only for the first TOTAL_TRANSACTIONS transactions and then returns. Before returning, the thread sets a global boolean variable, allTransactionsDone to true, indicating that no more paintings would be added to the inventory. By keeping TOTAL_TRANSACTIONS small (equal to 5), we were able to get just the right amount of transactions to demonstrate the concept of mutex -- not too less, not too more!

The threadPaintingsOut stops when allTrasactionsDone becomes true and when it is done selling all the remaining paintings. On the other hand, if there are no paintings and allTrasactionsDone is still false, then the thread unlocks the mutex, sleeps for some time, and then retries.

The program intentionally keeps different values of the maximum time for sleep for the two threads. For threadPaintingsIn, the max value is MAX_SLEEP_SECONDS_PAINTINGS_IN seconds, where as for threadPaintingsOut, the max value is MAX_SLEEP_SECONDS_PAINTINGS_OUT seconds. By keeping different values, we create a case where the threadPaintingsOut wakes up more often and checks if there is anything in the inventory.

Lastly, if you are feeling uncomfortable with the varying levels of indentation (the tabs "\t" in the printf() statements), then, well it is also intentional! We have kept it to improve readability of the output: the events of the main thread have no tabs, the events when paintings arrive in the inventory have a single tab, and the events when paintings are sold have two tabs.

To run the above example, we compile it with "-pthread" option to add support for Pthreads library. Here is the output:

```
 [user@codingtree]$ gcc mutex_two_threads.c -pthread -o mutex-two-threads
 [user@codingtree]$
 [user@codingtree]$ ./mutex-two-threads
 Waiting for the threadPaintingsIn..
        [paintingsIn]Sleep for 2 seconds.
```

```
                [paintingsOut]Sleep for 2 seconds.
        [paintingsIn]Added one painting. Total paintings are: 1
        [paintingsIn]Sleep for 9 seconds.
                [paintingsOut]Sold one painting. Total paintings left: 0
                [paintingsOut]Sleep for 3 seconds.
                [paintingsOut]No more paintings left. Let us retry.
                [paintingsOut]Sleep for 0 seconds.
                [paintingsOut]No more paintings left. Let us retry.
                [paintingsOut]Sleep for 2 seconds.
                [paintingsOut]No more paintings left. Let us retry.
                [paintingsOut]Sleep for 3 seconds.
                [paintingsOut]No more paintings left. Let us retry.
                [paintingsOut]Sleep for 3 seconds.
        [paintingsIn]Added one painting. Total paintings are: 1
        [paintingsIn]Sleep for 5 seconds.
                [paintingsOut]Sold one painting. Total paintings left: 0
                [paintingsOut]Sleep for 4 seconds.
        [paintingsIn]Added one painting. Total paintings are: 1
        [paintingsIn]Sleep for 8 seconds.
                [paintingsOut]Sold one painting. Total paintings left: 0
                [paintingsOut]Sleep for 2 seconds.
                [paintingsOut]No more paintings left. Let us retry.
                [paintingsOut]Sleep for 2 seconds.
                [paintingsOut]No more paintings left. Let us retry.
                [paintingsOut]Sleep for 1 seconds.
                [paintingsOut]No more paintings left. Let us retry.
                [paintingsOut]Sleep for 1 seconds.
                [paintingsOut]No more paintings left. Let us retry.
                [paintingsOut]Sleep for 3 seconds.
        [paintingsIn]Added one painting. Total paintings are: 1
        [paintingsIn]Sleep for 0 seconds.
        [paintingsIn]Added one painting. Total paintings are: 2
 Waiting for the threadPaintingsOut..
                [paintingsOut]Sold one painting. Total paintings left: 1
                [paintingsOut]Sleep for 1 seconds.
                [paintingsOut]Sold one painting. Total paintings left: 0
```

It is easy to see that both threads take turns to do their work and do not overwrite the common variable. In other words, they are synchronized. The threadPaintingsOut does (intentionally) wake up more often and if threadPaintingsIn did not add any paintings, then threadPaintingsOut unlocks the mutex and then later.

The output shows that this may not be optimal because threadPaintingsOut has to retry multiple times before there is some painting that needs to be sold. It would have been more efficient, if threadPaintingsOut were to wait till threadPaintingsIn adds a new painting. But, how does threadPaintingsOut know when threadPaintingsIn has added a new painting? One common way for the threadPaintingsIn to tell threadPaintingsOut that it has added a painting is to use Pthread condvars.

POSIX Threads: Conditional Variables (Condvars)

Mutexes allow us to achieve synchronization, that is among a set of threads, who gets to access the critical data first. But, this is just one side of the Pthread story. For cases where the thread needs to wait for some common data, how does it know how long to wait? For such cases, threads need to communicate with each other.

Let us consider a simple case of two threads. Let us also say that the first thread produces a data and the second thread consumes that data. It is possible that when the second thread is trying to consume data, there is no data and so it must wait. Without explicit communication, the second thread would have to wait and check periodically if the resource has become available. Imagine kids sitting in the car on a road-trip and asking continuously, "Are we there yet?".

For such cases, the second thread should ideally wait till the first thread is done producing the resource. Once done, the first thread can communicate to the second thread to go ahead. Using an explicit communication for such use-cases is more efficient.

Pthreads achieve this inter-thread communication using conditional variables (or condvars for short). Condvars work hand in hand with mutexes. Pthreads provides a new variable type for condvars as well: pthread_cond_t. Thus, if we have to define a Pthread condvar variable, x, then we can do it as "pthread_cond_t x".

First of all, let us begin by putting together some of the common Pthread condvar APIs. Here they are:

```
 int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
 int pthread_cond_timedwait(pthread_cond_t *c, pthread_mutex_t *m, struct
timespec *t);
 int pthread_cond_signal(pthread_cond_t *c);
 int pthread_cond_broadcast(pthread_cond_t *c);
 int pthread_cond_init(pthread_cond_t *c, pthread_condattr_t *attr);
 int pthread_cond_destroy(pthread_cod_t *c);
```

The first API in the above list, pthread_cond_wait() allows us to wait on a condvar; it takes both a condvar and a mutex as parameters. The caller must lock the mutex before calling pthread_cond_wait(), otherwise, the behavior is undefined! If the pthread_cond_wait() is successful, then it immediately unlocks the mutex and the then blocks the current thread. The idea behind unlocking the mutex is some other thread (hopefully, the thread for which we are waiting!) can acquire the mutex and continue its work.

Function pthread_cond_timedwait() is same as pthread_cond_wait(), except that it takes a time value as a parameter; if the time of wait exceeds the specified time value, then pthread_cond_timedwait() returns with an error of ETIMEDOUT. Threads blocked on pthread_cond_wait() and pthread_cond_timedwait() are often referred to as waiter threads.

The next API, pthread_cond_signal(), helps a thread signal the waiter threads and tell that it has provided the resource for which they are waiting. So now, a waiter thread can wake up and continue their task. As soon as a waiter thread receives this signal, pthread_cond_wait() unblocks and with unblocking, it immediately acquires the mutex. Once the thread is done with its task, it

should call pthread_mutex_unlock() to release the mutex. Note that the Pthread signal is not the same as Linux/Unix signals (like SIGKILL, SIGTERM, etc) -- these two types are unrelated.

Instead of pthread_cond_signal(), a thread can also call pthread_cond_broadcast(), if there are more than one waiter threads. Using pthread_cond_broadcast() is more efficient since it informs all the waiter threads in one shot. If multiple threads are waiting, then the thread that gets the lock (remember, pthread_cond_wait() returns and implicitly acquires the lock) depends upon the scheduling policy. In essence, it is same as multiple threads vying to lock a single mutex. Those who fail to lock the mutex must continue to play the waiting game!

Needless to say, if there are no waiter threads, then pthread_cond_signal() and pthread_cond_broadcast() do not do anything.

Like the case of mutexes, the last two APIs allow us to initialize and destroy a condvar dynamically. We can use pthread_cond_init() to initialize a condvar variable. Once done, we can use pthread_cond_destroy() to release resources attached with it.

It is also possible to define (and initialize) a condvar variable statically (meaning that the scope is only the current file) using the PTHREAD_COND_INITIALIZER macro; this macro contains a default values for various attributes of a condvar. Thus, if want to define a condvar statically, then we can do that as: "pthread_cond_t x = PTHREAD_COND_INITIALIZER;". For statically defined condvars, there is no need to call pthread_cond_destroy().

With that, let us provide an example, where two threads, threadPaintingsIn and threadPaintingsOut, use a condvar to communicate with each other. The thread threadPaintingsIn mimics buying of paintings from artists by incrementing a global variable, totalPaintings. The thread threadPaintingsOut mimics selling of paintings to end-users by decrementing the same global variable, totalPaintings.

If there are no paintings, then threadPaintingsOut uses pthread_cond_wait() to simply wait. The threadPaintingsIn sleeps for a random interval and upon waking up, adds a new painting. After it adds a new painting, it uses pthread_cond_signal() to signal the waiter thread.

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <stdbool.h>

#define TOTAL_TRANSACTIONS 5
#define MAX_SLEEP          5

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;

static int totalPaintings = 0;
static bool allTransactionsDone = false;
static bool thread_waiting = false;

/* Sell the paintings from the inventory */
```

```
void *paintingsOut (void *arg) {
    int random_time;

    while ((allTransactionsDone == false) || totalPaintings) {
        pthread_mutex_lock(&mutex);
        if (!totalPaintings) {
            thread_waiting = true;
            printf("\t\t[%s]No more paintings left. Let us wait..\n",
__FUNCTION__);
            pthread_cond_wait(&condvar, &mutex);
            thread_waiting = false;
        }

        totalPaintings--;
        printf("\t\t[%s]Sold one painting. Total paintings left: %d\n",
__FUNCTION__, totalPaintings);
        pthread_mutex_unlock(&mutex);
    }
}

/* Buy painting from artists and add to the Inventory */
void *paintingsIn (void *arg) {
    int i, random_time;

    srand(time(NULL));
    for (i = 0; i < TOTAL_TRANSACTIONS; i++) {
        random_time = rand() % MAX_SLEEP;
        printf("\t[%s]Sleep for %d seconds. \n", __FUNCTION__, random_time);
        sleep(random_time);

        pthread_mutex_lock(&mutex);
        totalPaintings++;
        printf("\t[%s]Added one painting. Total paintings are: %d\n",
                    __FUNCTION__, totalPaintings);
        pthread_mutex_unlock(&mutex);
        if (thread_waiting) {
            pthread_cond_signal(&condvar);
        }
    }
    allTransactionsDone = true;
}

int main () {
    pthread_t threadPaintingsIn, threadPaintingsOut;
    int status;

    status = pthread_create(&threadPaintingsIn, NULL, paintingsIn, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_create() failed [status: %d]\n", status);
        return 0;
    }

    status = pthread_create(&threadPaintingsOut, NULL, paintingsOut, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_create() failed [status: %d]\n", status);
        return 0;
    }
```

```
    printf("Waiting for the threadPaintingsIn..\n");
    status = pthread_join(threadPaintingsIn, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_join() failed for threadPaintingsIn
[%d]\n", status);
    }

    printf("Waiting for the threadPaintingsOut..\n");
    status = pthread_join(threadPaintingsOut, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_join() failed for threadPaintingsOut
[%d]\n", status);
    }
    return 0;
 }
```

Some additional notes on the above program.

For simplicity sake, the threadPaintingsIn does not run forever. It runs only for the first TOTAL_TRANSACTIONS transactions and then returns. Before returning, the thread sets a global boolean variable, allTransactionsDone to true, indicating that no more paintings would be added to the inventory.

The threadPaintingsOut stops when allTrasactionsDone becomes true and when it is done selling all the remaining paintings. On the other hand, if there are no paintings and allTrasactionsDone is still false, then it uses pthread_cond_wait() to wait till threadPaintingsIn signals it to wake up.

Lastly, we add varying levels of indentation (the tabs "\t" in the printf() statements) to improve readability of the output: the events of the main thread have no tabs, the events when paintings arrive in the inventory have a single tab, and the events when paintings are sold have two tabs.

To run the above example, we compile it with "-pthread" option that adds support for Pthreads library. The output shows that with condvar, threadPaintingsOut behaves patiently -- if there is no painting, then it waits on the condvar and tries only when it receives a signal from threadPaintingsIn.

```
 [user@codingtree]$ gcc condvar_two_threads.c -pthread -o condvar-two-threads
 [user@codingtree]$
 [user@codingtree]$ ./condvar-two-threads
 Waiting for the threadPaintingsIn..
                [paintingsOut]No more paintings left. Let us wait..
        [paintingsIn]Sleep for 0 seconds.
        [paintingsIn]Added one painting. Total paintings are: 1
        [paintingsIn]Sleep for 2 seconds.
                [paintingsOut]Sold one painting. Total paintings left: 0
                [paintingsOut]No more paintings left. Let us wait..
        [paintingsIn]Added one painting. Total paintings are: 1
        [paintingsIn]Sleep for 2 seconds.
                [paintingsOut]Sold one painting. Total paintings left: 0
                [paintingsOut]No more paintings left. Let us wait..
        [paintingsIn]Added one painting. Total paintings are: 1
```

```
        [paintingsIn]Sleep for 1 seconds.
                [paintingsOut]Sold one painting. Total paintings left: 0
                [paintingsOut]No more paintings left. Let us wait..
        [paintingsIn]Added one painting. Total paintings are: 1
        [paintingsIn]Sleep for 1 seconds.
                [paintingsOut]Sold one painting. Total paintings left: 0
                [paintingsOut]No more paintings left. Let us wait..
        [paintingsIn]Added one painting. Total paintings are: 1
Waiting for the threadPaintingsOut..
                [paintingsOut]Sold one painting. Total paintings left: 0
[user@codingtree]$
```

POSIX Threads: Deadlocks

One of the pain points of having multiple threads is dealing with deadlocks. A deadlock occurs when we have a circular dependency between threads and resources; these resources can be as simple as Pthread mutexes. Deadlocks are bad because the threads involved in the deadlock do nothing but sit there forever. Not exactly what we had in mind, when we created them!

For example, let us say we have a thread, (say t1) and that is holding a resource (say r1). But as per the application logic, t1 also needs to acquire another resource (say r2). Now as luck would have it, r2 is being held by a second thread (say t2) and that is actually waiting for r1. Thus, t1 is waiting for r2 which is held by t2, which in turn, is waiting for r1 that is held by t1! Due to this, threads t1 and t2 would wait indefinitely. The following picture illustrates this deadlock.

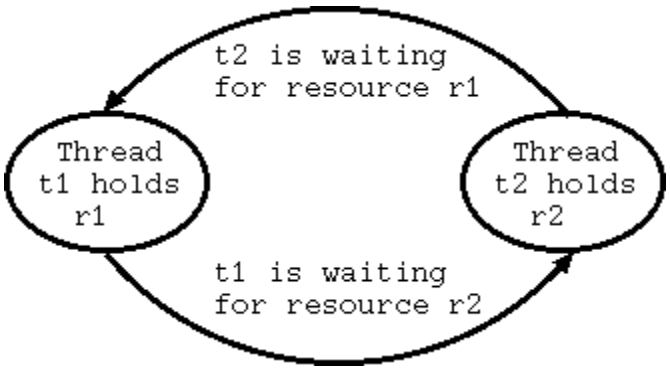


Figure: Deadlock between two threads (t1 and t2)

Having two threads wait for each other is the simplest case for a deadlock. It is possible that we can have N threads run into a deadlock such that each one is waiting for a resource that is needed by the other thread. Think of this as a loop. The first thread waits for a resource held by the second thread, the second thread waits for a resource held by third thread, and the last thread waits for the resource held by the first thread.

Having threads wait for a resource in a circular fashion is at the heart of a deadlock. Further, for deadlocks to happen, two or more resources should be such that they cannot be shared among multiple threads. If we take a mutex as a resource, then this sounds right because a mutex can

have only one owner and the thread that locks it is the only thread that can unlock it. The ownership is one of the key reasons why we end up with deadlocks.

Investigating deadlocks

Debugging a deadlock when dealing with multiple threads might be one of the hardest things a programmer would ever have to do! When threads decide to sit in a deadlock, refusing to move an inch further, you can forget about relying upon debugs. With threads sitting in a blocked state, they would not execute any statements, and so there is absolutely no way, they would print any debugs.

When dealing with a deadlock, one way to start debugging is to consider all possible paths, where threads can go and thereby deadlock. Clearly, when we have scores of threads, this method can have its own limitation. For such cases, one approach is to fire GDB and when threads sit in a stuck state, look at the state of each thread, and each resource (e.g. mutex).

With that goal in mind, we provide an example that artificially creates a deadlock and then debug it using GDB. The example (provided below) has a single goal of creating a deadlock. It would be safe to say that this example does not do anything even remotely useful!

The example has two threads, kept in an array thread[]. Each thread has its own callback function; callback1() and callback2(). The example uses two mutexes as resources: mutex1 and mutex2.

The callback functions are similar in nature that they both sleep for random times and when they wake up, they try to lock both mutexes, one after another. However, the ordering of the mutexes is different. The first callback attempts to lock mutex1 first, followed by mutex2. The second callback attempts to lock mutex2 first, followed by mutex1. Due to this, the first thread locks mutex1 and goes to sleep. The second thread locks mutex2 and goes to sleep. When thread1 wakes up, it tries to lock mutex2, but that is already being held by thread2. Likewise, when thread2 wakes up, it tries to lock mutex1, which is being held by thread1. In other words, we get into a deadlock!

Each of these callback functions sit within a while loop. The reason for that is it is possible that the two threads could miss the deadlock in the first pass. This can happen if one of the threads sleep longer than the two combined sleep of the other thread. Let us say, thread1 sleeps for 1 second and after locking, it sleeps for 2 seconds. If thread2 were to sleep for 3 second, then it would wake up exactly when thread1 is done in the first pass. So, if the threads miss the deadlock, having a loop forces them to retry achieving deadlock in subsequent passes!

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>

#define MAX_SLEEP 10
#define MAX_THREADS 2

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *callback1 (void *arg) {
    int random_time;

    while (1) {
        srand(time(NULL));
        random_time = rand() % MAX_SLEEP;
        printf("[%s]Sleeping for %d seconds\n", __FUNCTION__, random_time);
        sleep(random_time);
        printf("[%s]Trying to acquire mutex1 (holding none)\n",
__FUNCTION__);
        pthread_mutex_lock(&mutex1);
        printf("[%s]Acquired mutex1\n", __FUNCTION__);

        random_time = rand() % MAX_SLEEP;
        printf("[%s]Sleeping for %d seconds\n", __FUNCTION__, random_time);
        sleep(random_time);

        printf("[%s]Trying to acquire mutex2 (holding mutex1) \n",
__FUNCTION__);
        pthread_mutex_lock(&mutex2);
        printf("[%s]Acquired mutex2\n\n", __FUNCTION__);

        pthread_mutex_unlock(&mutex2);
        pthread_mutex_unlock(&mutex1);
    }
    return NULL;
}

void *callback2 (void *arg) {
    int random_time;

    while (1) {
        srand(time(NULL));
        random_time = rand() % MAX_SLEEP;
        printf("[%s]Sleeping for %d seconds\n", __FUNCTION__, random_time);
        sleep(random_time);
        printf("[%s]Trying to acquire mutex2 (holding none)\n",
__FUNCTION__);
        pthread_mutex_lock(&mutex2);
        printf("[%s]Acquired mutex2\n", __FUNCTION__);

        random_time = rand() % MAX_SLEEP;
        printf("[%s]Sleeping for %d seconds\n", __FUNCTION__, random_time);
        sleep(random_time);

        printf("[%s]Trying to acquire mutex1 (holding mutex2) \n",
__FUNCTION__);
        pthread_mutex_lock(&mutex1);
        printf("[%s]Acquired mutex1\n\n", __FUNCTION__);

        pthread_mutex_unlock(&mutex1);
        pthread_mutex_unlock(&mutex2);
    }
    return NULL;
}
```

```
int main () {
    pthread_t thread[MAX_THREADS];
    int status, i;

    status = pthread_create(&thread[0], NULL, callback1, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_create failed [status: %d]\n", status);
        return -1;
    }

    status = pthread_create(&thread[1], NULL, callback2, NULL);
    if (status != 0) {
        fprintf(stderr, "pthread_create failed [status: %d]\n", status);
        return -1;
    }

    for (i=0; i < MAX_THREADS; i++) {
        status = pthread_join(thread[i], NULL);
        if (status != 0) {
            fprintf(stderr, "pthread_join failed for thread1 [%d]\n",
status);
            return -1;
        }
    }
    return 0;
}
```

Let us now compile and run the program. We use the pthread option with gcc to add support for Pthreads library.

```
[codingtree]$ gcc -pthread deadlock_two_threads.c  -o deadlock
[codingtree]$
[codingtree]$ ./deadlock
[callback1]Sleeping for 4 seconds
[callback2]Sleeping for 4 seconds
[callback1]Trying to acquire mutex1 (holding none)
[callback1]Acquired mutex1
[callback1]Sleeping for 2 seconds
[callback2]Trying to acquire mutex2 (holding none)
[callback2]Acquired mutex2
[callback2]Sleeping for 9 seconds
[callback1]Trying to acquire mutex2 (holding mutex1)
[callback2]Trying to acquire mutex1 (holding mutex2)


^C
[codingtree]$
```

The output confirms what we expected. The thread1 sleeps for 4 seconds, wakes up and locks mutex1. Then it sleeps for 2 seconds again and when wakes up, tries to lock mutex2. The thread2 has a similar story. After the last line in the output, both threads sit in a deadlock and if program is not killed (using Control-C), then they would sit there waiting for mutexes, for ever!

We can use GDB to look into the states of these threads and mutexes. For that, let us recompile the program by passing "-g" option to gcc; this option enables gcc to build symbols which are later used by GDB. We provide the output first and explain it after that.

```
 [codingtree]$ gcc -g -pthread deadlock_two_threads.c  -o deadlock
 [codingtree]$
 [codingtree]$ gdb deadlock
 GNU gdb (GDB) Fedora (7.3.50.20110722-16.fc16)
 Copyright (C) 2011 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.  Type "show copying"
 and "show warranty" for details.
 This GDB was configured as "i686-redhat-linux-gnu".
 For bug reporting instructions, please see:
 <http://www.gnu.org/software/gdb/bugs/>...
 Reading symbols from /home/codingtree/threads/deadlock...done.
 (gdb)
 (gdb) run
 Starting program: /home/codingtree/threads/deadlock
 [Thread debugging using libthread_db enabled]
 Using host libthread_db library "/lib/libthread_db.so.1".
 [New Thread 0xb7fe8b40 (LWP 4786)]
 [callback1]Sleeping for 7 seconds
 [New Thread 0xb77e7b40 (LWP 4787)]
 [callback2]Sleeping for 7 seconds
 [callback1]Trying to acquire mutex1 (holding none)
 [callback1]Acquired mutex1
 [callback1]Sleeping for 6 seconds
 [callback2]Trying to acquire mutex2 (holding none)
 [callback2]Acquired mutex2
 [callback2]Sleeping for 2 seconds
 [callback2]Trying to acquire mutex1 (holding mutex2)
 [callback1]Trying to acquire mutex2 (holding mutex1)

 ^C
 Program received signal SIGINT, Interrupt.
 0xb7fff424 in __kernel_vsyscall ()
 Missing separate debuginfos, use: debuginfo-install glibc-2.14.90-
24.fc16.9.i686
 (gdb)
 (gdb) info threads
   Id   Target Id         Frame
   3    Thread 0xb77e7b40 (LWP 4787) "deadlock" 0xb7fff424 in
__kernel_vsyscall ()
   2    Thread 0xb7fe8b40 (LWP 4786) "deadlock" 0xb7fff424 in
__kernel_vsyscall ()
 * 1    Thread 0xb7fe9900 (LWP 4783) "deadlock" 0xb7fff424 in
__kernel_vsyscall ()
 (gdb)
 (gdb) thread apply all bt full

 Thread 3 (Thread 0xb77e7b40 (LWP 4787)):
 #0  0xb7fff424 in __kernel_vsyscall ()
 #1  0x450f34d2 in __lll_lock_wait () from /lib/libpthread.so.0
```

```
#2  0x450eee63 in _L_lock_693 () from /lib/libpthread.so.0
#3  0x450eeca8 in pthread_mutex_lock () from /lib/libpthread.so.0
#4  0x080488ae in callback2 (arg=0x0) at deadlock_two_threads.c:54
        random_time = 2
        __FUNCTION__ = "callback2"
#5  0x450eccd3 in start_thread () from /lib/libpthread.so.0
#6  0x45029d7e in clone () from /lib/libc.so.6

Thread 2 (Thread 0xb7fe8b40 (LWP 4786)):
#0  0xb7fff424 in __kernel_vsyscall ()
#1  0x450f34d2 in __lll_lock_wait () from /lib/libpthread.so.0
#2  0x450eee63 in _L_lock_693 () from /lib/libpthread.so.0
#3  0x450eeca8 in pthread_mutex_lock () from /lib/libpthread.so.0
#4  0x08048750 in callback1 (arg=0x0) at deadlock_two_threads.c:28
        random_time = 6
        __FUNCTION__ = "callback1"
#5  0x450eccd3 in start_thread () from /lib/libpthread.so.0
#6  0x45029d7e in clone () from /lib/libc.so.6

Thread 1 (Thread 0xb7fe9900 (LWP 4783)):
#0  0xb7fff424 in __kernel_vsyscall ()
#1  0x450edde5 in pthread_join () from /lib/libpthread.so.0
#2  0x080489b9 in main () at deadlock_two_threads.c:80
        thread = {3086912320, 3078519616}
        status = 0
        i = 0
(gdb)
(gdb) thread 3
[Switching to thread 3 (Thread 0xb77e7b40 (LWP 4787))]
#0  0xb7fff424 in __kernel_vsyscall ()
(gdb)
(gdb) print mutex1
$1 = {__data = {__lock = 2, __count = 0, __owner = 4786, __kind = 0,
        __nusers = 1, {__spins = 0, __list = {__next = 0x0}}},
        __size = "\002\000\000\000\000\000\000\262\022\000\000\000\000\
        000\000\001\000\000\000\000\000\000", __align = 2}
(gdb)
(gdb) print mutex2
$2 = {__data = {__lock = 2, __count = 0, __owner = 4787, __kind = 0,
        __nusers = 1, {__spins = 0, __list = {__next = 0x0}}},
        __size = "\002\000\000\000\000\000\000\263\022\000\000\000\000\
        000\000\001\000\000\000\000\000\000", __align = 2}
(gdb)
(gdb) info threads
  Id   Target Id         Frame
* 3    Thread 0xb77e7b40 (LWP 4787) "deadlock" 0xb7fff424 in
__kernel_vsyscall ()
  2    Thread 0xb7fe8b40 (LWP 4786) "deadlock" 0xb7fff424 in
__kernel_vsyscall ()
  1    Thread 0xb7fe9900 (LWP 4783) "deadlock" 0xb7fff424 in
__kernel_vsyscall ()
(gdb)
(gdb) bt
#0  0xb7fff424 in __kernel_vsyscall ()
#1  0x450f34d2 in __lll_lock_wait () from /lib/libpthread.so.0
#2  0x450eee63 in _L_lock_693 () from /lib/libpthread.so.0
#3  0x450eeca8 in pthread_mutex_lock () from /lib/libpthread.so.0
```

```
#4  0x080488ae in callback2 (arg=0x0) at deadlock_two_threads.c:54
#5  0x450eccd3 in start_thread () from /lib/libpthread.so.0
#6  0x45029d7e in clone () from /lib/libc.so.6
(gdb)
(gdb) continue
Continuing.
```

First, we need to run the program with GDB as "gdb deadlock" -- once we hit the GDB prompt, we enter "run" to run the program under GDB. After sometime, we would hit the deadlock and the program would halt. At that point, we can use Control-C to get into GDB prompt. Don't worry, even with Control-C, the GDB is still running the program and to get back to program, all we need to do is enter "continue"!

Once we get into GDB prompt, we can use "info threads" to see all the threads belonging to a program. In the output, each of the threads has an LWP id: thread 1 (LWP id 4783), thread 2 (LWP id 4786), and thread 3 (LWP id 4787); LWP stands for Light Weight Process. We can also see this from the GDB log that threads 2 and 3 were created from the main thread, when we issued the "run" command: "[New Thread 0xb7fe8b40 (LWP 4786)]" and "[New Thread 0xb77e7b40 (LWP 4787)]".

The asterisk (*) in the "info threads" output shows us the current thread in the GDB context. If needed, we can go from one thread to another by specifying the thread number to the "thread" command. Thus, "thread 2" would take us into the context of thread 2.

The next command "thread apply all bt full", basically prints backtrace of all threads; "bt" is short for backtrace. So, when we run it, we see backtrace of all the three threads. It shows that thread 1 started in the main(), thread 2 started in callback1(), and thread 3 started in callback2(). It also shows that two of the threads (2 and 3) are waiting on the pthread_mutex_lock() -- this should be our first hint (but not definitive hint) that we might have a deadlock.

We should also make a note of backtrace number 4 for threads 2 and 3. For thread 2, it says " #4 0x08048750 in callback1 (arg=0x0) at deadlock_two_threads.c:28", which means that it is stuck at line number 28, where we are calling the pthread_mutex_lock for mutex2. Similarly, for thread 3, it says "4 0x080488ae in callback2 (arg=0x0) at deadlock_two_threads.c:54", which means that it is stuck at line number 54, where we are calling the pthread_mutex_lock for mutex1. Another potent hint that we might be approaching the forbidden deadlock zone!

The last hint comes when we switch to thread 3 and print values of mutex1 and mutex2. When we print mutex1, we see that it has an owner with an LWP value of 4786 and LWP 4786 represents thread 2. Along the same lines, when we print mutex2, we see that it has an owner with an LWP value of 4787 and LWP 4787 represents thread 3. It is this data point that confirms we have a deadlock. From the traceback, we see that thread 2 is waiting for mutex2 and from printing the mutex2, we know that mutex2 is owned by thread 3. Similarly, from the traceback, we see that thread 3 is waiting for mutex1 and from printing the mutex, we know that mutex1 is owned by thread 2. Hence, the deadlock!

Fixing deadlocks

Finding the root cause of a deadlock is an important step towards fixing the deadlock. But, once we know that a deadlock exists, we also need to provide a fix for it. The fix varies from case to case. Let us discuss some of the methods that can help us avoid deadlock or to break an ongoing deadlock.

First, for cases where multiple resources (e.g. mutexes) are to be allocated for a given thread, we should ensure that the resource-acquisition happens in a pre-specified order. For our above example, specifying the order could mean once a thread has acquired the first mutex, then it is the same thread that will also acquire the second mutex. In this way, the second thread would not wait for the second mutex since it needs to acquire the first mutex first.

Needless to say that if we have certain threads that need to acquire only one of the resource during their life-time, then they can continue doing so without having to bother acquiring all the threads. So, enforcing acquisition of resources in a predefined order should be enforced only on those threads that need to acquire multiple resources.

Second, for applications where it is possible to know all the resources needed by a thread in advance, we can also use what is known as Banker's algorithm to avoid deadlocks. This algorithm does a book-keeping of resources and whenever it allocates a resource, it checks whether it can likely cause a deadlock.

Third, we do not have to wait indefinitely for a resource. For code-locations, where trying to acquire a resource would likely lead to a deadlock, we can replace the calls with pthread_mutex_trylock() and if it fails because the mutex is already locked, then we can retry later. If the call still fails, then that might be a good hint that we might be hitting a deadlock. In that case, we can release the resources held by the thread and either return from the call or resume the call later.

In fact, if the deadlock occurs rarely, then depending upon the application logic, it might be reasonable to simply return with an error instead of resuming the call. For example, let us say that we have two threads that are trying to send socket data. If the two threads run into a deadlock, then with the above approach, one of them can simply return without sending data. This would probably be okay for applications that can handle a rare packet loss.

Lastly, it is also common for some implementations to have a watch dog. A watch dog can be a thread that runs independently and monitors all the other threads. If certain threads are blocked for a long period of time, then the watchdog can detect that and conclude that there might be a deadlock.

Socket Programming: Introduction

Sockets are entry-points for a network "pipe" that connects two or more network applications. Both sides need to open a socket to create the network "pipe". Sockets are bidirectional in nature and hence, both ends of the "pipe" can simultaneous send data to the other end and receive data from the other end. The two endpoints typically sit on different machines. However, there is nothing stopping us from making them sit on the same machine.

The following figure shows two sockets forming a network "pipe".

Figure: Communication using two sockets

Technically speaking, sockets are a part of GNU C library (also known as glibc). We include them here since a lot of network applications are popularly written in C using glibc sockets.

A socket is identified by three main parameters: (a) IP address (IPv4 or IPv6) of the machine, (b) the port number on that machine, and (c) the transport protocol. A port number is a logical local identification point for network applications; each host can have as much as 65,536 ports. In terms of network stack, transport protocols (e.g. TCP or UDP) belong to a layer that sits on top of the IP layer but beneath the application layer. Sockets on both sides need to use the same transport protocol.

Depending upon the transport protocol, sockets can communicate with each other in two ways: connection-oriented or connection-less. In addition to transport sockets, we also have sockets that allow us to send and receive raw packets. We do not cover those sockets here.

Connection-oriented Sockets

For connection-oriented sockets, the main thing is that the two sockets must establish an explicit connection (hand-shake, if you will) before they can send or receive data. The primary transport protocol for these types of sockets is Transmission Control Protocol (TCP).

To setup a connection, we need to create a server socket that can wait for connection requests from (remote) clients. Then, we need to create a client socket that can send a request. Once the server receives a request from a client, it creates a new connection. To be more precise, with each request, the server socket creates a new local socket and associates it with the remote client socket. This way, the two sockets -- the client that initiated the connection request and the one that was created locally by the server socket -- become connected.

Once a connection is established, the parent server socket does not call it quits. Instead, it detaches itself from the new connection and starts waiting for the next incoming request. And the cycle continues.

The following figure shows connection establishment in two steps. In the first step, the server socket (socketA0) accepts an incoming connection from a client socket (socketB0). In the second

step, the server socket creates a new local socket (socketA1) and associates it with the the client socket. With that, socketA1 and socketB0 now become connected.

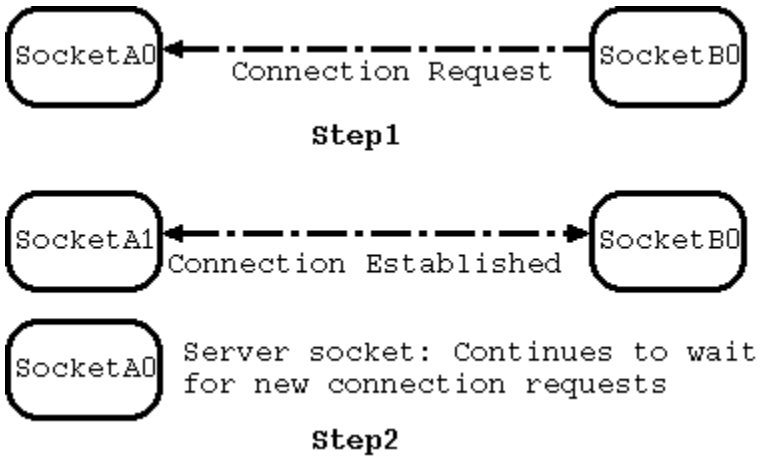
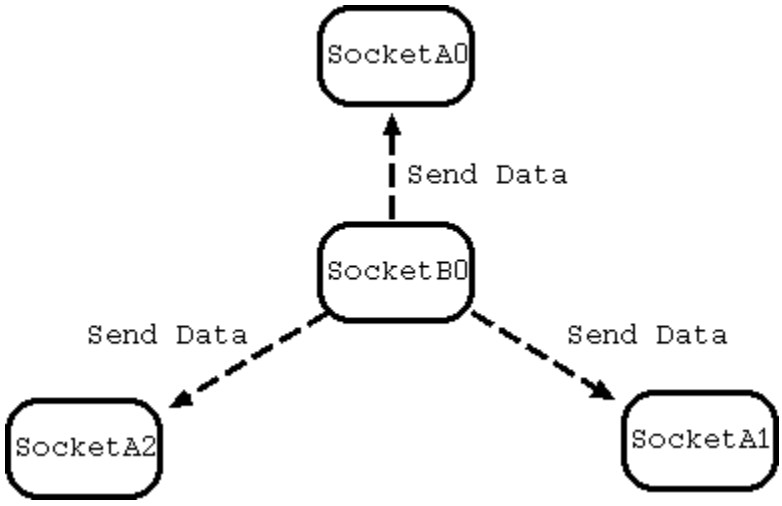


Figure: Connection Establishment for TCP sockets

Connectionless Sockets

For connectionless sockets, there is no requirement for an explicit connection setup. A client socket can send messages directly to a server socket. The primary transport protocol for these types of sockets is User Datagram Protocol (UDP). Because such sockets are connectionless, one socket can send messages to multiple server sockets at the same time -- something that is not possible with the connection-oriented sockets.

The following figure shows a connectionless client socket (socketB0) sending data to three different server sockets (socketA0, socketA1, and socketA2), all at the same time.



FigureSocket Programming: TCP Sockets (Connection-oriented Sockets)

TCP sockets are connection-oriented sockets and hence, require an explicit association between the two socket endpoints. Two connected sockets learn the address of each other during the connection-setup and so they can send data to each other without having to specify the address of the other socket.

Having an explicit connection enables TCP to offer three important flow-properties. First, if the network drops packets, then TCP retransmits lost packets. Second, TCP adjusts the sending-rate, when the available network bandwidth changes. Lastly, TCP sender ensures that it does not send more data than the available receiver buffer at the peer (the other endpoint) and thus, does not overwhelm the receiver. If these services are critical for your application, then you should consider using TCP.

Depending upon who initiates the connection, a socket can be classified as a server socket or as a client socket. The server socket accepts incoming requests for new connections from client sockets. The client socket sends request for a new connection. This section provides APIs that allow us to create both types of sockets. Following that, we provide their sample implementation as well.

TCP Socket Server

First off, let us discuss socket APIs that allow us to build a TCP server socket. Let us start with signature of these APIs (provided below), followed by their discussion and an implementation.

```
int socket(int family, int type, int protocol);
int bind(int fd, const struct sockaddr *addr, socklen_t addrlen);
int listen(int fd, int backlog);
int accept(int fd, struct sockaddr *addr, socklen_t *addrlen);
int close(int fd);
```

The socket() call creates a socket -- whether it is a server or a client -- and is usually the very first function in a socket application. This call accepts three parameters. The first parameter represents the IP address family that can be AF_INET (for IPv4 family of addresses), AF_INET6 (for IPv6 family of addresses), AF_UNIX (for communicating with sockets on the local machine), and AF_PACKET(for sending/receiving packets directly to network driver without having to go through the TCP/IP (or UDP/IP) stack). The second parameter is the protocol type that can be SOCK_STREAM (for TCP) or SOCK_DGRAM (for UDP). The last parameter is the transport protocol that can be IPPROTO_TCP (for TCP) or IPPROTO_UDP (for UDP). Thus, if we want to create an IPv4 TCP socket, then these params would be AF_INET4, SOCK_STREAM, and IPPROTO_TCP. For IPv6, they would be AF_INET6, SOCK_STREAM, and IPPROTO_TCP.

The return value of the socket() call is a file descriptor that is used to symbolically refer to this socket. All subsequent operations on this socket (like send or receive) must be done by passing this file descriptor. If this call runs into an error, then it returns a value of -1.

The next three functions collectively characterize the steps needed to create a TCP server: bind(), listen(), and accept(). The bind() call allows the server to bind to a well-known port and an address so that clients can reach it. The listen() call allows the server to wait (or listen) for incoming connections from clients. Once a request arrives, the accept() call allows the server to retrieve a new connection.

Let us look at these three calls in more detail.

The bind() call takes three params: (a) the file descriptor of the socket that we need to bind, (b) a pointer to an address structure that holds the local port and local IP address, and (c) the length of the address buffer pointed by the address pointer. Upon success, the bind() call returns 0, else it returns -1.

Some thoughts on port numbers. Each host can have as many as 65,536 ports for each protocol and for each address family; typically, port numbers in the range of 0 to 1024 are standard ports and are reserved for various applications. Ports outside this range are usually available for general use. For a detailed list of well-known ports, please visit Internet Assignment Numbers Authority (IANA) website.

Binding establishes the server socket uniquely in the entire Internet. Any remote socket client can send connection request to this server using a combination of port, machine's IP address, and the protocol. The only requirement is that the combination of these three variables should be unique in the Internet. If we were to use the analogy of regular postage mail, then the bind() call would mean assigning (or using) a unique mailing address to a house. It is this uniqueness that allows the post office to deliver all the mails destined to this house correctly.

The next call, listen() takes two parameters: (a) the file descriptor of the socket that needs to become a listener socket and (b) the maximum number of backlog of pending connections. This call allows a socket to wait for newer connections from remote client machines. Once a socket is in listen mode and it receives a request for a connection, then it completes TCP's 3-way handshake and enqueues the new connection in a queue that is reserved for pending connections. For our postal analogy, the listen() call means that the owner of the house puts a mailbox so that the postman can deliver incoming mails. Upon success, the listen() call returns 0, else it returns -1.

The backlog limit is a good strategy to avoid Denial-Of-Service attacks. Without this limit, a malicious client can continuously (and at a fast rate) send connection requests. Since each connection requires both memory and CPU processing, this attack can seriously deplete memory and CPU resources at the server.

Even though having a backlog limit is a good thing, it is possible that the server can get busy and thus, can take more time to dequeue newer connections from the accepted list. For such cases,

some of the genuine requests can also be dropped since the queue of pending connections might already be at the maximum backlog limit! We should not loose our sleep over this because TCP has an inbuilt mechanism where the clients would retry again and hopefully, with the next try, the server would have dequeued some of the pending connections.

One last thing about the backlog. The backlog value is bounded by an upper limit provided by the underlying operating system. For Linux, the default value is 256. Thus, if an application were to erroneously pass this parameter as a high value, let us say 1000, then the underlying layer would automatically reduce it to 256.

The next step of accept() takes three params: (a) the file descriptor of the server socket, (b) a pointer to an address structure, and (c) the length of the storage pointed by the address pointer. Upon success, the accept() call returns file descriptor of the new (child) socket, else it returns -1.

The accept() call allows the application to retrieve a single connection from the queue of pending connections. The retrieval dequeues the first connection from the queue and hence, creates room for one additional future connection. This call returns a new file descriptor associated with the new connection. This file descriptor is different from that of the server and all socket operations for the new connection should be done using this descriptor. The accept() call is blocking because if there is no pending connection, then the calling thread must wait.

Even though we do not need to bind to any local address, the accept() call still takes "struct sockaddr" buffer as argument. The reason for passing an address buffer is that the accept call returns the address of the remote client (or peer) that initiates the connection. This way, the server application gets to know the IP address and the port number of the remote client socket. Not bad!

On error, all of these calls set the system errno variable, besides returning -1. We should make it a habit of printing the value of the error number, when we run into an error. The error information would prove handy during difficult times of debugging!

The following figure displays the above three steps of bind(), listen(), and accept() for server socket (socketA0) and its interaction with a client socket (socketB0); the client socket uses a connect() call and we will discuss that a little later. Note that the steps for client side may happen at the same machine.

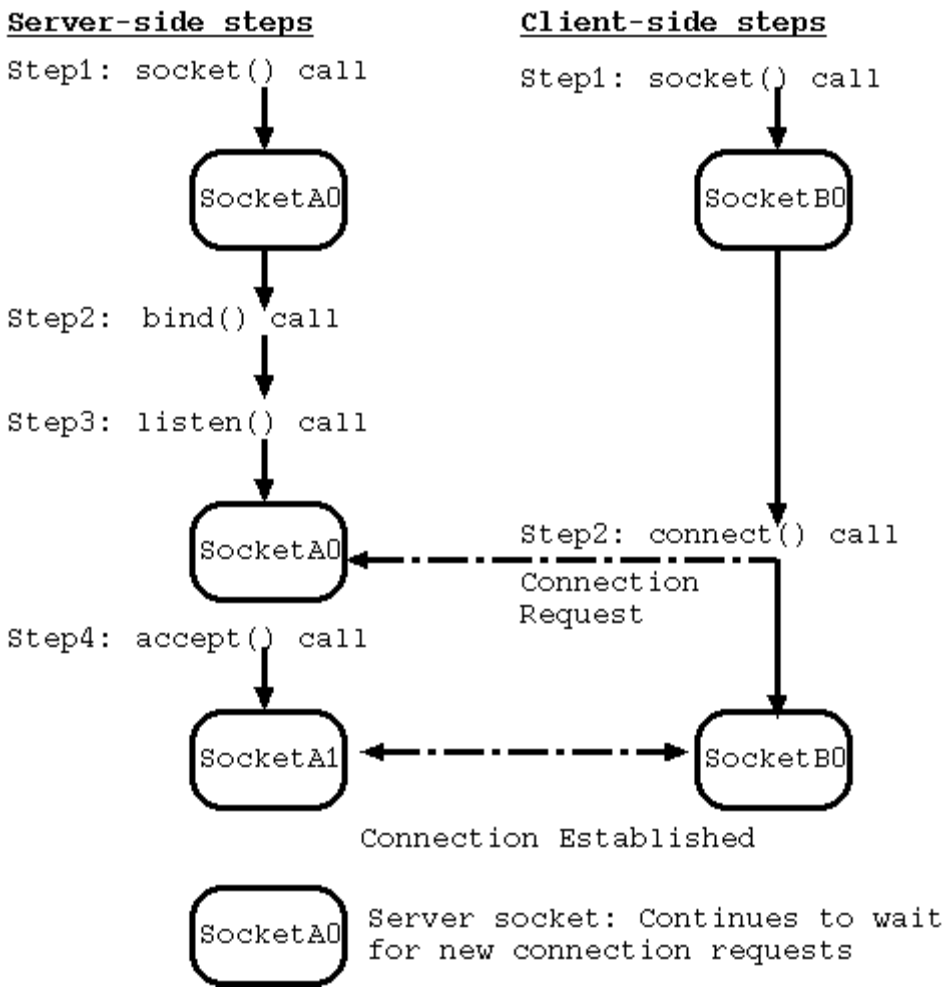


Figure: Detailed Steps for TCP Connection Establishment

Once we are done with all the operations, we can call close() and pass the file descriptor of the socket that we wish to close. This call is essential since it releases all the resources held by the socket. For TCP, this call also informs the other end (the peer) that it is tearing down the connection. On success it returns 0, else it returns -1.

While the earlier calls have focused on creating a socket and setting up the connection, the next two calls allow a socket server to send and receive data. A client socket also uses the same functions to send and receive data, so these calls are actually common to both server and client sockets. Since sockets are bidirectional, both sides can send data and receive data simultaneously.

```
ssize_t send(int fd, const void *buf, size_t bufsize, int flags);
ssize_t recv(int fd, void *buf, size_t bufsiz, int flags);
```

The send() call sends data to the other end of the connected socket pipe. This call takes three parameters: (a) the file descriptor of the sending socket, (b) a pointer to a buffer that contains the data we wish to send, and (c) the length of the buffer storage. The underlying socket/TCP layer

copies this data into the outgoing send buffer. Upon success, this call returns the number of bytes sent and on error, it returns -1.

If the buffer is more than the space available in the send buffer, then the send() call can also block; the underlying socket/TCP layer maintains its own send buffer for each socket. However, if the socket is non-blocking, then it would return -1 with the errno set to EAGAIN or EWOULDBLOCK.

The recv() call allows an application to read received data. It takes three parameters: (a) the file descriptor of the server socket, (b) a pointer to a buffer, and (c) the length of the buffer storage. We pass buffer so that the underlying socket/TCP layer can copy received data (from the client) into this buffer. Upon success, this call would return the number of bytes received and copied in the buffer. On error it would return -1. As a special case, if recv() call returns zero, then that means the peer has (gracefully) closed the connection.

The recv() call is a blocking call. If there is no received data, then the calling thread will block till it receives any data or till the underlying connection is closed; like the send buffer, the underlying socket/TCP layer also maintains its own receiver buffer for every socket. If the socket is non-blocking and if there is no received data, then recv() would return with -1 and the errno will be set to EAGAIN or EWOULDBLOCK. In that case, the application would have to retry later.

Depending upon the amount of data received by the TCP layer (let us say, k_rcvd), the passed buffer (let us say, k_passed) may or may not be sufficient. If the passed buffer is more than the data received (i.e. k_passed > k_rcvd), then the TCP layer will return all the data received in the same buffer and the returned value would be k_rcvd. On the other hand, if the passed buffer is less than the data received (i.e. k_passed < k_rcvd), then TCP will only return k_passed bytes.

The recv() call also accepts a flag that can be passed to tweak the behavior of the recv() call itself. Two of these flags are: MSG_DONTWAIT and MSG_PEEK. MSG_DONTWAIT specifies that if the underlying TCP has not received any data, then it can return immediately and the returned value would be -1. MSG_PEEK means that the normal recv() call behavior would hold except that the TCP layer would not remove that much data from its receive buffer since the goal is only to peek; we would need a subsequent recv() call without MSG_PEEK flag set to drain the data from the TCP receive buffer.

That completes our rather-long discussion on socket functions needed for a server. If you feel like grabbing a cup of coffee, I would understand! Let us move on and apply our skills to write a simple server. We present the example below and following that, we describe its various pieces.

```
#include <stdio.h>
#include <errno.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define DATA_BUFFER 5000

int create_tcp_server_socket() {
```

```
    struct sockaddr_in saddr;
    int fd, ret_val;

    /* Step1: create a TCP socket */
    fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (fd == -1) {
        fprintf(stderr, "socket failed [%s]\n", strerror(errno));
        close(fd);
        return -1;
    }
    printf("Created a socket with fd: %d\n", fd);

    /* Initialize the socket address structure */
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(7000);
    saddr.sin_addr.s_addr = INADDR_ANY;

    /* Step2: bind the socket to port 7000 on the local host */
    ret_val = bind(fd, (struct sockaddr *)&saddr, sizeof(struct
sockaddr_in));
    if (ret_val != 0) {
        fprintf(stderr, "bind failed [%s]\n", strerror(errno));
        close(fd);
        return -1;
    }

    /* Step3: listen for incoming connections */
    ret_val = listen(fd, 5);
    if (ret_val != 0) {
        fprintf(stderr, "listen failed [%s]\n", strerror(errno));
        close(fd);
        return -1;
    }
    return fd;
}

int main () {
    struct sockaddr_in new_client_addr;
    int fd, new_fd, ret_val;
    socklen_t addrlen;
    char buf[DATA_BUFFER];

    /* Create the server socket */
    fd = create_tcp_server_socket();
    if (fd == -1) {
        fprintf(stderr, "Creating server failed [%s]\n", strerror(errno));
        return -1;
    }

    /* Accept a new connection */
    new_fd = accept(fd, (struct sockaddr*)&new_client_addr, &addrlen);
    if (new_fd == -1) {
        fprintf(stderr, "accept failed [%s]\n", strerror(errno));
        close(fd);
        return -1;
    }
    printf("Accepted a new connection with fd: %d\n", new_fd);
```

```
    /* Receive data */
    printf("Let us wait for the client to send some data\n");
    do {
        ret_val = recv(new_fd, buf, DATA_BUFFER, 0);
        printf("Received data (len %d bytes)\n", ret_val);
        if (ret_val > 0)
            printf("Received data: %s\n", buf);
        if (ret_val == -1) {
            printf("recv() failed [%s]\n", strerror(errno));
            break;
        }
    } while (ret_val != 0);

    /* Close the sockets */
    close(fd);
    close(new_fd);
    return 0;
}
```

Now, let us describe various pieces of the above example.

The example includes "netinet/in.h" and "sys/socket.h" for definitions of socket address types, constants, and socket calls.

Next, the main() function calls the create_tcp_server_socket() function to create a TCP server socket. This function begins by using socket() call to create a TCP socket.

The create_tcp_server_socket() function binds the socket using a bind() call to a unique port number (7000). The function htons() converts the unsigned ushort port number (port numbers are defined as unsigned short) from host byte order to network byte order. This is needed to handle different endianness of machines. For address, it passes INADDR_ANY (equal to zero) that means the default local address on the server.

Following the bind() call, the create_tcp_server_socket() function uses listen() call to make the server wait for incoming calls; it passes a backlog of 5. After that, it returns with the file descriptor of the server socket.

The main() issues an accept() call using the server file descriptor and waits for an incoming connection. Once the server receives an incoming request, the accept() call returns a new file descriptor (new_fd).

We should take a moment to note that the bind() and accept() calls take pointer to "struct sockaddr" instead of "struct sockaddr_in". The reason for this is that "struct sockaddr" is an IPv4/IPv6 independent definition. For IPv6, we need to use "struct sockaddr_in6" as the socket address. However, for both sockaddr_in and sockaddr_in6, the first 2 bytes (which is the address family) is common and that is same as the first 2 bytes of sockaddr. Thus, irrespective of sockaddr_in or sockaddr_in6, the first 2 bytes are bound to the address family.

```
struct sockaddr_in {
```

```
    short            sin_family;   // AF_INET
    unsigned short   sin_port;     // port number in network byte order
    struct in_addr   sin_addr;     // IP address.
    char             sin_zero[8];  // zero this if you want to
};
struct sockaddr_in6 {
    u_int16_t       sin6_family;   // AF_INET6
    u_int16_t       sin6_port;     // port number in network byte order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};
struct sockaddr {
    unsigned short    sa_family;    // AF_INET or AF_INET6
    char              sa_data[14];  // Protocol address
};
```

When the bind() call receive a socket address, it uses the first 2 bytes to identify the address family. If the address family is AF_INET, then it uses sockaddr_in to interpret the rest of the fields. On the other hand, if the address family is AF_INET6, then it uses sockaddr_in6 to interpret the rest of the fields. Thus, this simple little trick allows us to retain the same function signature for both IPv4 and IPv6 families!

To keep our implementation simple, the server waits for only one connection. After a new connection is established, the server waits for data on the new connection using the recv() call; else, why would we go through all this trouble! Since recv() call returns zero when the connection is closed, we stay in the do-while loop as long as the return value of the recv() call is not zero. Once recv() call returns 0 bytes, we call it a day, close both sockets, and go home!

TCP Socket Client

Having described the implementation of the TCP server program, let us now look at the other side of the story. Where as a TCP server waits for new connections, a TCP client is the one that actually issues these requests.

In terms of functions, some of the calls used by a client are same as that of the server socket: socket(), send(), recv(), and close(). Since they are already discussed above, we omit their discussion here.

However, the one call that differentiates a client from the server is a connect() call -- it is this call that allows the client to send a request to the server and thereby establish a new connection. Like the bind() call, the connect() call also takes an address as a parameter. However, the passed address is that of the remote TCP server. Here is its signature:

```
int connect(int fd, const struct sockaddr *addr, socklen_t addrlen);
```

With that, let us write a simple TCP client program.

```
#include <stdio.h>
#include <errno.h>
```

```
 #include <netinet/in.h>
 #include <netdb.h>

 #define DATA_BUFFER "Mona Lisa was painted by Leonardo da Vinci"

 int main () {
     struct sockaddr_in saddr;
     int fd, ret_val;
     struct hostent *local_host; /* need netdb.h for this */

     /* Step1: create a TCP socket */
     fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
     if (fd == -1) {
         fprintf(stderr, "socket failed [%s]\n", strerror(errno));
         return -1;
     }
     printf("Created a socket with fd: %d\n", fd);

     /* Let us initialize the server address structure */
     saddr.sin_family = AF_INET;
     saddr.sin_port = htons(7000);
     local_host = gethostbyname("127.0.0.1");
     saddr.sin_addr = *((struct in_addr *)local_host->h_addr);

     /* Step2: connect to the TCP server socket */
     ret_val = connect(fd, (struct sockaddr *)&saddr, sizeof(struct
sockaddr_in));
     if (ret_val == -1) {
         fprintf(stderr, "connect failed [%s]\n", strerror(errno));
         close(fd);
         return -1;
     }
     printf("The Socket is now connected\n");

     printf("Let us sleep before we start sending data\n");
     sleep(5);

     /* Next step: send some data */
     ret_val = send(fd,DATA_BUFFER, sizeof(DATA_BUFFER), 0);
     printf("Successfully sent data (len %d bytes): %s\n", ret_val,
DATA_BUFFER);

     /* Last step: close the socket */
     close(fd);
     return 0;
 }
```

For the sake of simplicity, we create this socket on the same machine as that of the TCP server. Therefore, we use the loopback address of the localhost (127.0.0.1) for the connect() call. If the client were to run on a different machine, then this call would require the IP address of the box that houses the TCP server socket.

You may have noticed that the bind() call is conspicuously absent from the scene for the client! This is because, we don't need to explicitly bind the socket to do a connect(). If the socket is not bound, then during the connect() step, the underlying TCP layer automatically selects a random

(available) port number and binds the client socket to it. This is acceptable since the client socket does not need to be bound to a well-known (or a pre-communicated) port number.

After connection is established, we use the send() call to send some data to the server. As explained before, since this is a connection-oriented socket, we do not have to specify the address of the remote client for every send() call! After sending data, we close the socket.

Making them talk!

Now that we have both the programs ready, let us run them together! For this task, we run the server on one terminal and the client on another. Further, we need to run the server first, since the server must be up and running before the client can send its request. Following that, we can run the client.

We provide below the output for the server. In the output, the message indicating the return of the accept call appears only after we start the client. Note that the file descriptor for the server socket is 3 where as the file descriptor for the newly accepted connection is 4. In the end, we close the client socket and accordingly, the server's recv() call returns with 0 bytes, indicating that the client has closed the connection.

```
 $ gcc tcp-server.c -o server
 $
 $ ./server
 Created a socket with fd: 3
 Accepted a new connection with fd: 4
 Let us wait for the client to send some data
 Received data (len 43 bytes)
 Received data: Mona Lisa was painted by Leonardo da Vinci
 Received data (len 0 bytes)
```

Following is the output for the client.

```
 $ gcc tcp-client.c -o client
 $
 $ ./client
 Created a socket with fd: 3
 The Socket is now connected
 Let us sleep before we start sending data
 Successfully sent data (len 43 bytes): Mona Lisa was painted by Leonardo da
Vinci
```

When debugging sockets, we can check the state of these sockets using the ss tool on Linux or netstat tool on non-Linux (Windows, or Mac OS, etc) machines. The netstat tool is deprecated on Linux. If we were to run these tools after running the "server" program and before running the client, we would find an entry for a TCP socket sitting on port 7000 and with a state of LISTEN. To have a compact output, we pass a set of options to ss tool or netstat tool: "t" for tcp only sockets, "p" for printing associated programs, "l" for printing only listening sockets, and "n" for printing numeric addresses.

For these output, to access names of all programs, we must be logged as a root. We provide the output both as the root user and as a non-root user. The output shows tcp-server listening on port 7000 (the third entry in the output).

```
 Output when logged as a non-root user
 [user@codingtree]$ ss -tpln
 Recv-Q Send-Q  Local Address:Port Peer Address:Port
 0      128     127.0.0.1:631      *:*
 0      128     :::631             :::*
 0      5       *:7000             *:*   users:(("server",30206,3))
 0      10      127.0.0.1:25       *:*
 [user@codingtree]$ ss -tpln

 Output when logged as the root
 [root@codingtree]# ss -tpln
 Recv-Q Send-Q  Local Address:Port Peer Address:Port
 0      128     127.0.0.1:631      *:*   users:(("cupsd",1532,12))
 0      128     :::631             :::*
users:(("cupsd",1532,4),("systemd",1,18))
 0      5       *:7000             *:*   users:(("server",30206,3))
 0      10      127.0.0.1:25       *:*   users:(("sendmail",29022,4))
 [root@codingtree]#
```

On non-Linux systems, here is the output with netstat tool. Once again, we need to login as root, if we wish to see all processes.

```
 [user@codingtree]$ netstat -tpln
 (Not all processes could be identified, non-owned process info
  will not be shown, you would have to be root to see it all.)
 Active Internet connections (only servers)
 Proto Recv-Q Send-Q Local Address  Foreign Address  State   PID/Program name
 tcp        0      0 127.0.0.1:631  0.0.0.0:*        LISTEN  -
 tcp        0      0 0.0.0.0:7000   0.0.0.0:*        LISTEN  30206/./server
 tcp        0      0 127.0.0.1:25   0.0.0.0:*        LISTEN  -
 tcp        0      0 :::631         :::*             LISTEN  -
 [user@codingtree]$
```

If we were to run this tool immediately after starting the client, such that the connection is established, then we can spot both the new connection and the old existing listener; the sleep command in the TCP client makes taking this output more convenient, else the client program would send data and too quickly for us to print the output.

This time, we pass an "a" option which lists all connections (including listen and established ones) instead of "l" option. We also filter the output (using a "grep" command) for port 7000 since we are only interested in connections that are based on port 7000. The output shows that when the server is waiting for incoming data and when it is reading incoming data, the earlier listener socket continues to stay in the listen state. In addition, we see two records of TCP sockets that are in established state. Since both the sockets are on the same box, the two entries are the two ends of the same connection. Here is the output with ss tool (we are logged in as root).

```
 [root@codingtree]# ss -tpan | grep 7000
```

```
 LISTEN  0      5     *:7000             *:*
users:(("server",30206,3))
 ESTAB   0      0     127.0.0.1:52425    127.0.0.1:7000
users:(("client",30272,3))
 ESTAB   0      0     127.0.0.1:7000     127.0.0.1:52425
users:(("server",30206,4))
 [root@codingtree]#
```

For the above output, the Recv-Q and the Send-Q (the second and third columns) show data in bytes. Thus, if we were to send 100 bytes from the client and if the server were to not call recv() call, then the Recv-Q would show 100 bytes for the established connection.

On non-Linux systems, here is the output with netstat tool:

```
 [root@codingtree]# netstat -tpan | grep 7000
 tcp        0      0 0.0.0.0:7000     0.0.0.0:*        LISTEN
30206/./server
 tcp        0      0 127.0.0.1:52425  127.0.0.1:7000   ESTABLISHED
30272/./client
 tcp        0      0 127.0.0.1:7000   127.0.0.1:52425  ESTABLISHED
30206/./server
 [root@codingtree]#
```

Socket Programming: UDP Sockets (Connectionless Sockets)

Connectionless sockets do not require any explicit connection setup before sending and receiving data. For a connectionless server, all we need to do is to create a socket and bind it a known port; with that, we can start receiving data. For a client, we do not even have to bother with the bind step, simply create a socket and start sending data!

However, a lack of explicit connection means that each time we wish to send data, we now need to specify the address of the remote (receiver) socket. This also means that we can use one socket to send data to multiple sockets; thus, we can potentially change the address to a new socket endpoint for every send call.

Connectionless sockets use UDP (User Datagram Protocol) protocol and are characterized by sockets of type SOCK_DGRAM. Unlike TCP, UDP neither retransmits lost packets nor adjusts its sending-rate. In this regard, UDP is a rather simple protocol. This simplicity is not without its merit! One of the attractive features of UDP is that since it does not need to retransmit lost packets nor does it do any connection setup, sending data incurs less delay. This lower delay makes UDP an appealing choice for delay-sensitive applications like audio and video.

Let us now describe implementation of UDP-based sockets. More specifically, we discuss socket APIs that allow us to build a UDP server and a UDP client.

UDP Socket Server

We begin our discussion of a UDP socket server by listing various socket APIs that are needed by a server. Following that, we also provide an implementation of the same.

```
int socket(int family, int type, int protocol);
int bind(int fd, const struct sockaddr *addr, socklen_t addrlen);
int close(int fd);
```

Like the case of TCP sockets, the socket() call is the very first call for a server, or for that matter, a client. This call accepts three parameters. The first parameter is the address family that can be AF_INET (for IPv4) or AF_INET6 (for IPv6). The second parameter is the protocol type that can be SOCK_STREAM (for TCP) or SOCK_DGRAM (for UDP). The last parameter is the protocol and its value can be IPPROTO_TCP (for TCP) and IPPROTO_UDP (for UDP).

Thus, if we want to create an IPv4 UDP socket, then the params should be: AF_INET4, SOCK_DGRAM, and IPPROTO_UDP. For IPv6, they should be: AF_INET6, SOCK_DGRAM, and IPPROTO_UDP.

The return value of the socket() call is a file descriptor that is used to symbolically refer to this socket. If we have to do any operation on this socket, then we must pass this file descriptor. On error, this call returns -1 and sets the system errno variable. For debugging purposes, we should print the error for failed cases.

Unlike a TCP socket server, a UDP socket server neither uses listen() nor accept() call. Instead, it simply relies on a bind() call. The bind() call allows an application to specify a port number and an address on the local machine. Each host can have as much as 65,536 ports for each protocol; typically, port numbers in the range of 0 to 1024 are standard ports and are reserved for various applications. Ports outside this range are usually available for general use. For a detailed list of well-known ports, please visit Internet Assignment Numbers Authority (IANA) website.

The bind() calls takes three params: (a) the file descriptor of the socket that we need to bind, (b) a pointer to an address structure that holds the local port and local IP address, and (c) the length of the address buffer pointed by the address pointer. If successful, it returns 0, otherwise, this call returns -1 and also sets the errno variable. Once again, we should print the error number for error cases.

Binding is needed because this uniquely establishes the combination of port number and address for the UDP server in the whole Internet. Due to that, a remote socket client can send data the server as long as it knows the server port and the machine's address.

If we were to use the analogy of regular postage mail, then the bind() call would mean assigning (or using) a mailing address to a house. This mailing address has to be unique (throughout the world!) and it is this uniqueness that allows the post office to deliver all the mails destined to this house correctly.

One we are done with all the operations, we should call close() and pass the file descriptor of the socket that we wish to close. We should not forget to close the socket since this step releases all

the resources held by the socket. On success it returns 0, else it returns -1 and sets the errno variable.

The next four functions allow a UDP socket server to send and receive data. A client socket can also uses these functions.

```
 ssize_t sendto(int fd, const void *buf, size_t len, int flags,
                    const struct sockaddr *receiver_addr, socklen_t
addrlen);
 ssize_t sendmsg(int fd, const struct msghdr *msg, int flags);
 ssize_t recvfrom(int fd, void *buf, size_t len, int flags,
                    struct sockaddr *sender_addr, socklen_t *addrlen);
 ssize_t recvmsg(int fd, struct msghdr *msg, int flags);
```

The first two calls sendto() and sendmsg() send data to the other end of the socket pipe; sending data is dependent upon application logic and any side can send data.

The sendto() calls takes six parameters. The first four are: (a) the file descriptor of the sending socket, (b) a pointer to a buffer that contains the data we wish to send, (c) number of bytes of data present in the buffer that we wish to send, and (d) a flag to modify the behavior of the sendto() call. The underlying socket/UDP layer copies this data into the outgoing send buffer; the underlying socket/UDP layer maintains a send buffer for every socket. The last two parameters allow the sender to pass the address and addrlen of the remote UDP receiver.

The sendmsg() call is similar to sendto() except that four of the sendto() parameters (buf, len, receiver_addr, and addrlen) are nicely tucked in a msghdr structure. Therefore, before we do anything else, let us look at the msghdr structure:

```
 struct msghdr {
     void         *msg_name;       /* Address of the other end */
     socklen_t     msg_namelen;    /* Size of address */
     struct iovec *msg_iov;        /* Scatter/gather array */
     size_t        msg_iovlen;     /* No of elements in msg_iov array */
     void         *msg_control;    /* Ancillary buffer */
     size_t        msg_controllen; /* Length of ancillary buffer */
     int           msg_flags;      /* flags on received message */
 };

 struct iovec {
     void         *iov_base;       /* Starting address */
     size_t        iov_len;        /* No of bytes at iov_base address */
 };
```

The msg_name/msg_namelen fields are equivalent of sendto() call's receiver_addr and addrlen.

The msg_iov is a scatter/gather array of struct iovec. Each element of the msg_iov array is a struct iovec and hence, contains a pointer to the start of a scatter data buffer and the length of each scatter data. Basically, instead of having one large buffer (like the sendto() call), we now have multiple smaller buffers scattered (in the form of iovecs) throughout the memory. The msg_iovlen field specifies the number of these iovec buffer elements. Thus, when handling

msghdr, we would need to run a loop and collect (gather) all the scattered data. The underlying layer gathers all of these buffer in a single data packet before sending it out.

The last three fields (msg_control and msg_controllen and msg_flags are used for the receive side and so, we will skip their discussion for now). Since both sendto() and sendmsg() calls send data to the peer socket, if this is your first time writing a socket program, it would be okay if you choose to use sendto() since that is the simpler of the two! Once you have mastered sockets, you can use either of these, as you see fit.

For both of these calls, if the data to be sent is more than the space available in the underlying send buffer, then they can block. However, if the socket is non-blocking, then these calls would return -1 and set errno to EAGAIN or EWOULDBLOCK; when that happens, the application would have to retry later. On success, both of these call return the number of bytes sent and on error, they return -1.

The next call, recvfrom() enables an application to get received data. Like the sendto() call, this call also takes six params. The first four are: (a) file descriptor of the socket, (b) a pointer to a buffer where the underlying layer will copy received data, (c) number of bytes in the buffer, and (d) a flag to modify the behavior of the recvfrom() call. We pass buffer so that the underlying socket/TCP layer can copy received data (from the client) into this buffer.

The last two parameters are sender_addr and addrlen -- recvfrom() copies the address of the remote client into the src_addr buffer. This allows the application to know about the peer sitting on the other end of the UDP pipe. It is okay to pass sender_addr as NULL. In that case, the underlying UDP protocol would not return the address of sender.

The function recvmsg() is akin to recvfrom() except that the parameters, buffer pointer, the length of the buffer, sender_addr, and addrlen are clubbed together within a msghdr data structure. Fields msg_name and msg_namelen correspond to the params sender_addr and addrlen of recvfrom() call. When the msg_name field is not NULL and the msg_namelen is not zero, the underlying layer copies the sender's address and the address length.

What is really unique about msghdr for recvmsg() is that one can get miscellaneous ancillary data about the incoming data by specifying the fields msg_control and msg_controllen. The control messages are of yet another structure type, cmsghdr that holds level and type of the ancillary information.

Upon success, both recvfrom() and recvmsg() return the number of bytes of the received message that they copied in the passed buffer. Note that UDP is a datagram protocol and so messages are received in the same size as sent by the sender. So, if we receive a big message (k_rcvd) and the caller passes a smaller length value (k_passed), then UDP layer copies that many bytes (k_passed) from the message and discards the remaining bytes.

Both recvfrom() and recvmsg() are blocking. If the underlying UDP layer has no received data from the peer, then both of these calls would simply wait till the UDP layer socket receives some

data. if the socket is non-blocking and if there is no received data, then both calls would return -1 and set the errno to EAGAIN or EWOULDBLOCK.

These two receive calls also accept a flag as a parameter. The flag field helps us fine-tune the behavior of these calls. Two of these flags are: MSG_DONTWAIT and MSG_PEEK. MSG_DONTWAIT specifies that if the underlying UDP has no data, then the calls should return immediately -- in that case, the returned value would be -1. With MSG_PEEK, these calls would return the data requested, but would not delete the data from the receive buffer since the goal is only to peek; we would need a subsequent recvfrom()/recvmsg() call with no MSG_PEEK flag to drain the data from the UDP receive buffer.

One last note. The calls, sendto(), sendmsg(), recvfrom(), and recvmsg() are not limited to connectionless sockets. We can call them even from a connection-oriented socket (e.g. TCP). When calling sendto() and sendmsg(), the underlying layer would ignore the sender's address and sender address length, since the socket is already connected and so it does not really need the peer's address.

With that, let us begin our implementation of a UDP server. We provide the example first and then explain its various pieces.

```
 #include <stdio.h>
 #include <errno.h>
 #include <netinet/in.h>

 #define DATA_BUFFER 5000

 int main () {
     struct sockaddr_in saddr, new_addr;
     int fd, ret_val;
     char buf[DATA_BUFFER];
     socklen_t addrlen;

     /* Step1: open a UDP socket */
     fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
     if (fd == -1) {
         fprintf(stderr, "socket failed [%s]\n", strerror(errno));
         return -1;
     }
     printf("Created a socket with fd: %d\n", fd);

     /* Initialize the socket address structure */
     saddr.sin_family = AF_INET;
     saddr.sin_port = htons(7000);
     saddr.sin_addr.s_addr = INADDR_ANY;

     /* Step2: bind the socket */
     ret_val = bind(fd, (struct sockaddr *)&saddr, sizeof(struct
sockaddr_in));
     if (ret_val != 0) {
         fprintf(stderr, "bind failed [%s]\n", strerror(errno));
         close(fd);
         return -1;
```

```
    }

    /* Step3: Start receiving data. */
    printf("Let us wait for a remote client to send some data\n");
    ret_val = recvfrom(fd, buf, DATA_BUFFER, 0,
                (struct sockaddr *)&new_addr, &addrlen);
    if (ret_val != -1) {
        printf("Received data (len %d bytes): %s\n", ret_val, buf);
    } else {
        printf("recvfrom() failed [%s]\n", strerror(errno));
    }

    /* Last step: close the socket */
    close(fd);
    return 0;
}
```

The above program begins by creating a UDP socket using socket() call. After creating the socket, the program binds it to a port (in this case, 7000) using the bind() call. For address, we pass INADDR_ANY (equal to zero) that means the default local address on the server. Once the UDP socket is bound, we issue recvfrom() call that waits for any incoming UDP datagram. To keep things simple, once we receive a data, we close the socket server by using a close() call.

UDP Socket Client

Now that we have the example for UDP server ready, let us look at its counterpart: a UDP client. For that, we provide an example that implements a simple UDP client.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <netinet/in.h>
#include <netdb.h>

#define DATA_BUFFER "Mona Lisa was painted by Leonardo da Vinci"

int main () {
    struct sockaddr_in saddr;
    int fd, ret_val;
    struct hostent *host; /* need netdb.h for this  */

    /* Step1: open a UDP socket */
    fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (fd == -1) {
        fprintf(stderr, "socket failed [%s]\n", strerror(errno));
        return -1;
    }
    printf("Created a socket with fd: %d\n", fd);

    /* Next, initialize the server address */
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(7000);
    host = gethostbyname("127.0.0.1");
    saddr.sin_addr = *((struct in_addr *)host->h_addr);
```

```
    /* Step2: send some data */
    ret_val = sendto(fd,DATA_BUFFER, strlen(DATA_BUFFER) + 1, 0,
        (struct sockaddr *)&saddr, sizeof(struct sockaddr_in));
    if (ret_val != -1) {
        printf("Successfully sent data (len %d bytes): %s\n", ret_val,
DATA_BUFFER);
    } else {
        printf("sendto() failed [%s]\n", strerror(errno));
    }

    /* Last step: close the socket */
    close(fd);
    return 0;
 }
```

The example begins by creating a UDP socket using socket() call. Following that, it calls sendto() to send some data. Since this socket is not a connection-oriented socket (UDP client does not use a connect() call), we do have to specify the address of the remote client for the sendto() call. In some ways, it is a trade-off!

To keep things simple, the example passes host address as the localhost since it runs both server and the client on the same machine. Of course, if it were to be created on a different machine, the sendto() step would require the IP address of the UDP server. Lastly, the client sends a single datagram as the UDP server is expecting only one datagram and then closes the socket.

Like the case of a TCP client, the UDP client also does not bother to do the bind() call because only one of the two endpoints need to sit at a well-known (or pre-communicated) port. If the other UDP server needs to communicate data to the client, then it can always retrieve the port and address of the client since the recvfrom() call provides an option to retrieve client's address. The server can use this information to send a reply back to the client. So, even though the client does not do an explicit bind, the server can still send data as long as the client sends the data first!

Making them talk!

Now that both the server and client examples are ready, let us run them together and have some fun! To see the client-server interaction, we need to run the server and the client on different terminals. The server should be run first so that it can wait for the client.

Here is the output from server. When we run the server, it creates a socket with file descriptor value as 3 and then starts to wait for incoming data. Once we start the client (on another terminal), the server prints the last output. Please note that we can run both the server and the client on the same terminal as well. For that, we can first run the server in the background mode as "./udp_server &" and then run the client.

```
$ gcc udp-server.c -o udp_server
$
$ ./udp_server
Created a socket with fd: 3
```

```
 Let us wait for a remote client to send some data
 Received data (len 43 bytes): Mona Lisa was painted by Leonardo da Vinci
```

Following is the output for the client. The client creates a socket (also with file descriptor value as 3) and then send data right away!

```
 $ gcc udp-client.c -o udp_client
 $.
 $ ./udp_client
 Created a socket with fd: 3
 Successfully sent data (len 43 bytes): Mona Lisa was painted by Leonardo da
Vinci
```

When debugging UDP sockets, we can use ss tool on Linux and netstat tool on non-Linux (Windows or Mac OS) boxes to get additional information. Please note that the "netstat" tool is deprecated on Linux. If we were to run these tools after running the "udp_server" program, then we would find an entry for the UDP server, sitting on port 7000 and with a state of UCONN. The netstat shows the "State" column as UCONN or empty because UDP is a connectionless protocol and hence, lacks states similar to those of TCP.

To have a crisp output, we pass a set of options to both ss and netstat tools: "u" for udp only sockets, "p" for printing associated programs, "a" for printing all connections, and "n" for printing numeric addresses. Also, for these output, to access names of all programs, we must be logged as a root, else the "p" option would not print the programs associated with these sockets.

As mentioned earlier, we need to run the udp_server first and then get these outputs since if we run the udp_client, then the udp_server code would receive data and close the socket. Once the udp_server closes the socket, we would not be able to see any entry for the socket!

Here is the output for both ss and netstat tools.

```
 [root@codingtree]# ss -upan
 State    Recv-Q Send-Q   Local Address:Port   Peer Address:Port
 UNCONN   0      0        *:48408              *:*
users:(("dhclient",22086,20))
 UNCONN   0      0        *:7000               *:*
users:(("udp_server",23994,3))
 UNCONN   0      0        *:68                 *:*
users:(("dhclient",22086,6))
 UNCONN   0      0        *:631                *:*
users:(("cupsd",1549,13))
 UNCONN   0      0        :::47798             :::*
users:(("dhclient",22086,21))
 [root@codingtree]#
 [root@codingtree]# netstat -upan
 Active Internet connections (servers and established)
 Proto Recv-Q Send-Q Local Address    Foreign Address  State  PID/Program
name
 udp        0      0 0.0.0.0:48408    0.0.0.0:*               22086/dhclient
 udp        0      0 0.0.0.0:7000     0.0.0.0:*
23994/./udp_server
 udp        0      0 0.0.0.0:68       0.0.0.0:*               22086/dhclient
```

```
udp        0      0 0.0.0.0:631      0.0.0.0:*                   1549/cupsd
udp        0      0 :::47798         :::*                        22086/dhclient
[root@codingtree]#
```

Socket Programming: Socket Select

A fair number of socket calls, like accept() and recv(), are blocking. This can pose a problem for real-life network applications, where a socket server needs to handle a large number of clients. It is easy to see that with large number of clients, we would end up blocking most of the time and hence, would hardly scale! The way around this problem is to use the socket select() call -- select() allows us to monitor a large number of sockets, all in one shot without having to block individually for each socket.

You could argue that scalability can also be reached by using a large number of threads, with some of them reading incoming data, some of the writing outgoing data, and one or more of them accepting incoming connections. While this certainly allows us to handle additional load, a select() call would still be a better choice since it can monitor a larger number (around 1024) sockets in one shot -- having 1024 threads would not be feasible on most of the platforms! However, the best bang for the buck would come when we combine these two approaches and use select() call with multiple threads. In this approach, we can have one thread blocking on the select() call and threads handling read and write operations identified by the select().

The select() method accepts as an input a bitmask (structure fd_set) where we set bits corresponding to the set of file descriptors. Do not worry -- the socket layer provides handy macros to set these bits and check for these bits, so we do not have to worry about dealing with bits at all. Isn't that a relief!

Once we pass an fd_set with bits set for each file descriptor, then the select() call monitors all of them. If there is an event on any of those descriptors, then it returns immediately and informs the application that a given fd has an event and the application can act accordingly. We can find the fd (or fds) with an event by checking if the corresponding bit is set for the passed file descriptors.

Before we go any further, this would be a good time to see the signature of the select() call.

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
    struct timeval *timeout);

struct timeval {
    long int tv_sec;
    long int tv_usec
};
```

The select() call takes several arguments. The first argument is the highest file descriptor plus one. Thus, if we pass two file descriptors with values 2 and 10, then the nfds parameter should be 10 + 1 or 11 and not 2. The maximum number of sockets supported by select() has an upper limit, represented by FD_SETSIZE (typically 1024). For simpler programs, passing FD_SETSIZE as nfds should be more than sufficient!.

The next three parameters represent the three different types of events monitored by the select(): read, write, and exception events. A read event means that for a given fd, there is either some data to be read (so the application can call recv()) or a new connection has been established (so the application can call accept()). A write event means that for a given fd, the local send buffer has become non-empty and the application can send more data. An exception event means that there is some exception event like receiving out-of-band data.

These three parameters are pointers to fd_set values, one for read, one for write, and the other for exception. An application does not necessarily have to pass all of these fd_sets. For example, if the application is only interested in monitoring read events, then it can pass only the read fd_set and pass the other two as NULL. The select calls monitors all the file descriptors specified in the three fd_set bitmasks.

The sixth and the last argument to select() is a timeout value in the form of a pointer to a timeval structure. The first field, tv_sec stores the number of whole seconds of elapsed time. The second field, tv_usec stores the rest of the elapsed time (a fraction of a second) in the form of microseconds. If we pass a NULL value to this field, then the select() waits indefinitely for events. Otherwise, if we make the select timeout after a certain time, then we need to pass a non-NULL value of timeval to it.

If a timeout does not occur and there are some events (read, write, or exception) on the file descriptors, then the return value from select() is the total number of file descriptors that are ready with read, write, or exception events. Also, when select() returns, it overwrites each of the three fd_sets with information about the descriptors that are ready for the corresponding operation. So, if we use select() in a loop, then before calling select(), we need to reset the fd_sets every time with the file descriptor that we wish to monitor.

With that, let us now write a simple TCP server code that demonstrates the need for a select() call. The example shows that select() can do several things seamlessly like, handling multiple existing connections, listening for newer connections, etc. We provide the example below followed by its explanation.

```
#include <stdio.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define DATA_BUFFER 5000
#define MAX_CONNECTIONS 10

int create_tcp_server_socket() {
    struct sockaddr_in saddr;
    int fd, ret_val;

    /* Step1: create a TCP socket */
    fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (fd == -1) {
        fprintf(stderr, "socket failed [%s]\n", strerror(errno));
        return -1;
```

```
    }
    printf("Created a socket with fd: %d\n", fd);

    /* Initialize the socket address structure */
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(7000);
    saddr.sin_addr.s_addr = INADDR_ANY;

    /* Step2: bind the socket to port 7000 on the local host */
    ret_val = bind(fd, (struct sockaddr *)&saddr, sizeof(struct
sockaddr_in));
    if (ret_val != 0) {
        fprintf(stderr, "bind failed [%s]\n", strerror(errno));
        close(fd);
        return -1;
    }

    /* Step3: listen for incoming connections */
    ret_val = listen(fd, 5);
    if (ret_val != 0) {
        fprintf(stderr, "listen failed [%s]\n", strerror(errno));
        close(fd);
        return -1;
    }
    return fd;
 }

 int main () {
    fd_set read_fd_set;
    struct sockaddr_in new_addr;
    int server_fd, new_fd, ret_val, i;
    socklen_t addrlen;
    char buf[DATA_BUFFER];
    int all_connections[MAX_CONNECTIONS];

    /* Get the socket server fd */
    server_fd = create_tcp_server_socket();
    if (server_fd == -1) {
      fprintf(stderr, "Failed to create a server\n");
      return -1;
    }

    /* Initialize all_connections and set the first entry to server fd */
    for (i=0;i < MAX_CONNECTIONS;i++) {
        all_connections[i] = -1;
    }
    all_connections[0] = server_fd;

    while (1) {
        FD_ZERO(&read_fd_set);
        /* Set the fd_set before passing it to the select call */
        for (i=0;i < MAX_CONNECTIONS;i++) {
            if (all_connections[i] >= 0) {
                FD_SET(all_connections[i], &read_fd_set);
            }
        }
```

```
        /* Invoke select() and then wait! */
        printf("\nUsing select() to listen for incoming events\n");
        ret_val = select(FD_SETSIZE, &read_fd_set, NULL, NULL, NULL);

        /* select() woke up. Identify the fd that has events */
        if (ret_val >= 0 ) {
            printf("Select returned with %d\n", ret_val);
            /* Check if the fd with event is the server fd */
            if (FD_ISSET(server_fd, &read_fd_set)) {
                /* accept the new connection */
                printf("Returned fd is %d (server's fd)\n", server_fd);
                new_fd = accept(server_fd, (struct sockaddr*)&new_addr,
&addrlen);
                if (new_fd >= 0) {
                    printf("Accepted a new connection with fd: %d\n",
new_fd);
                    for (i=0;i < MAX_CONNECTIONS;i++) {
                        if (all_connections[i] < 0) {
                            all_connections[i] = new_fd;
                            break;
                        }
                    }
                } else {
                    fprintf(stderr, "accept failed [%s]\n",
strerror(errno));
                }
                ret_val--;
                if (!ret_val) continue;
            }

            /* Check if the fd with event is a non-server fd */
            for (i=1;i < MAX_CONNECTIONS;i++) {
                if ((all_connections[i] > 0) &&
                    (FD_ISSET(all_connections[i], &read_fd_set))) {
                    /* read incoming data */
                    printf("Returned fd is %d [index, i: %d]\n",
all_connections[i], i);
                    ret_val = recv(all_connections[i], buf, DATA_BUFFER, 0);
                    if (ret_val == 0) {
                        printf("Closing connection for fd:%d\n",
all_connections[i]);
                        close(all_connections[i]);
                        all_connections[i] = -1; /* Connection is now closed
*/
                    }
                    if (ret_val > 0) {
                        printf("Received data (len %d bytes, fd: %d): %s\n",
                            ret_val, all_connections[i], buf);
                    }
                    if (ret_val == -1) {
                        printf("recv() failed for fd: %d [%s]\n",
                            all_connections[i], strerror(errno));
                        break;
                    }
                }
                ret_val--;
                if (!ret_val) continue;
```

```
            } /* for-loop */
        } /* (ret_val >= 0) */
    } /* while(1) */

    /* Last step: Close all the sockets */
    for (i=0;i < MAX_CONNECTIONS;i++) {
        if (all_connections[i] > 0) {
            close(all_connections[i]);
        }
    }
    return 0;
}
```

The above example uses an all_connections array to store information about various sockets. This is needed since we are dealing with multiple sockets. Next, the example creates a server socket (using the socket(), bind(), and listen() call sequence) by calling the create_tcp_server_socket() and then stores the returned file descriptor as the first element in the all_connections array. Further, as and when we get new incoming connections, we store their fds in this array as well.

To help identify empty slots in the all_connections array, we initialize it with -1. And as and when a connection goes away, we reset its index back to -1. We use all_connections array to set bits in read_fd_set value by using the FD_SET() macro. For the sake of simplicity, we pass an fd_set only for read events and NULL for write and exception events.

The above program passes NULL as a timeout value to select() call. However, if an application wishes to timeout, then it can define a timeout value as follows and then pass "&timeout" as the last parameter to the argument. For example, the following example sets the timeout value to 30.5 seconds.

```
struct timeval timeout;
timeout.tv_sec = 30;      /* Value in seconds */
timeout.tv_usec = 500000; /* Value in milli-seconds */
```

The program sits in a while() loop and for every pass, it begins by populating the read_fd_set with the connections present in the all_connections array. Next, the program blocks on select() and the waiting game begins! Once we have an incoming connection or some data on an existing connection, the select() returns. Since upon return(), the select() call updates the read_fd_set by first clearing all the bits and then setting only those bits that have read events. We use FD_ISSET macro to find out the connections that have been set.

For a listener fd, which is the first element of the all_connections array, a read-event signifies that there is a pending new connection. And, when that happens, we can call accept() and store the returned file descriptor of the new connection in the all_connections array. Since the return value of select() is the total number of file descriptors that are ready for the event, we decrease the ret_val value by one. If ret_val equals zero, then that means only one file descriptor was ready and so we continue.

Since all_connections can have two or more file descriptors (one listener and the other accepted connections), we should be able to listen to incoming read events for all of them -- a read event on the listener would mean a new connection and a read event on the accepted connection would mean there is new data to be read.

For a non-listener fd, if the received bytes is 0, then that means the connection is closed and we remove it from the all_connections array so that in the next pass, we would not be doing a select() for the closed connection.

It would be good to pinpoint that the above program uses two loops: one to traverse over the array and set each file descriptor in the read_fd_set and one to lookup the returned file descriptor from the select() call. Each of the two loops incur a running time complexity of O(n) -- clearly, we can do better than that. One way to improve the running time complexity would be to use a hash-table so that the lookup loop becomes faster. The average time complexity of a hash table is O(1) and so, for higher-workloads, it would be a lot faster. If we want to make both the loops faster, then we would have to use a data structure that is optimized for both traversal and lookups. We would leave that as an exercise for the reader!

Now that we have the server ready, we would also need clients that can talk to it and test our select() code. For that we present a simple TCP client program. Once we write it, we would run multiple copies of this client to mimic a workload of multiple clients.

```
 #include <stdio.h>
 #include <errno.h>
 #include <netinet/in.h>
 #include <netdb.h>

 #define DATA_BUFFER "Mona Lisa was painted by Leonardo da Vinci"

 int main () {
     struct sockaddr_in saddr;
     int fd, ret_val;
     struct hostent *local_host; /* need netdb.h for this */

     /* Step1: create a TCP socket */
     fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
     if (fd == -1) {
         fprintf(stderr, "socket failed [%s]\n", strerror(errno));
         return -1;
     }
     printf("Created a socket with fd: %d\n", fd);

     /* Let us initialize the server address structure */
     saddr.sin_family = AF_INET;
     saddr.sin_port = htons(7000);
     local_host = gethostbyname("127.0.0.1");
     saddr.sin_addr = *((struct in_addr *)local_host->h_addr);

     /* Step2: connect to the TCP server socket */
     ret_val = connect(fd, (struct sockaddr *)&saddr, sizeof(struct
sockaddr_in));
     if (ret_val == -1) {
```

```
        fprintf(stderr, "connect failed [%s]\n", strerror(errno));
        close(fd);
        return -1;
    }
    printf("The Socket is now connected\n");

    printf("Let us sleep before we start sending data\n");
    sleep(5);

    /* Next step: send some data */
    ret_val = send(fd,DATA_BUFFER, sizeof(DATA_BUFFER), 0);
    printf("Successfully sent data (len %d bytes): %s\n",
                ret_val, DATA_BUFFER);

    /* Last step: close the socket */
    close(fd);
    return 0;
}
```

This time we would run two clients from two different terminals along with this server. The output for the server is below.

```
$ gcc select-tcp-server.c -o select_tcp_server
$
$ ./select_tcp_server
Created a socket with fd: 3

Using select() to listen for incoming events
Select returned with 1
Returned fd is 3 (server's fd)
Accepted a new connection with fd: 4

Using select() to listen for incoming events
Select returned with 1
Returned fd is 3 (server's fd)
Accepted a new connection with fd: 5

Using select() to listen for incoming events
Select returned with 1
Returned fd is 4 [index, i: 1]
Received data (len 43 bytes, fd: 4): Mona Lisa was painted by Leonardo da
Vinci

Using select() to listen for incoming events
Select returned with 1
Returned fd is 4 [index, i: 1]
Closing connection for fd:4

Using select() to listen for incoming events
Select returned with 1
Returned fd is 5 [index, i: 2]
Received data (len 43 bytes, fd: 5): Mona Lisa was painted by Leonardo da
Vinci

Using select() to listen for incoming events
Select returned with 1
```

```
 Returned fd is 5 [index, i: 2]
 Closing connection for fd:5

 Using select() to listen for incoming events

 ^C
 $
```

From this output, we can see that a listener is created with a file descriptor of 3 and it then uses select() to wait for incoming connections. When the first client comes up, the select() returns and we accept the first incoming connection assigning it a file descriptor of 4. After that we do select() and block, only to be awakened by the next client's incoming connection which gets an fd of 5. At this point, we call select by setting 3 fds in the fd_set: 3, 4,and 5. When we receive data on the new connections, select() returns and we read data. Once a connection is closed, select() returns (when the recv() call returns 0) and we close the connection. And, the select() will continue to wait forever for the newer connections.

The output for both the clients is same:

```
 $ ./tcp_client
 Created a socket with fd: 3
 The Socket is now connected
 Let us sleep before we start sending data
 Successfully sent data (len 43 bytes): Mona Lisa was painted by Leonardo da
Vinci
```

We should note that this example uses select() for a connection-oriented socket (TCP), but select() can also be used for connectionless sockets (UDP). If there are multiple UDP listeners (servers), then we can always use select() to block for all of these and as and when we receive a read event, we can use recvfrom() to read the incoming data.

Socket Programming: Socket Options

Sockets are versatile in terms of options they support. We can use these socket options as knobs to customize the socket behavior as per the needs of a network application. For example, if an application sends data in occasional large bursts, we can use a socket option to increase the size of the socket send buffer, so that it can such bursts.

Sockets support several options. Socket layer provides two functions that help us set/get these options:

```
 int setsockopt(int fd, int level, int name, const void *value, socklen_t
len);
 int getsockopt(int fd, int level, int name, const void *value, socklen_t
*len);
```

The first argument for both calls is the file descriptor of the socket for which we wish to set or get the option.

The second and third arguments for both calls specify the level and the option name for the option. Each socket option is identified by the combination of an option name and an option level; the level identifies the layer in the stack for which the behavior is being modified. For example, if we wish to modify socket layer behavior, then we can set the level to reflect the socket layer.

The last two arguments for the setsockopt() call are the value and the length of the option. The value is a pointer and the storage represented by the pointer is indicated by the option_len parameter. When we call setsockopt(), the application sets the option_value along with its length and passes it to the underlying layer.

Once an option is set for a socket, we can use getsockopt() call to retrieve its value. With getsockopt(), we pass an empty buffer (in the form of value parameter) along with the length of the buffer. This time, the underlying socket layer sets the option value in the buffer. Once getsockopt() returns, the application can read the value of the option from the buffer.

Sockets support a host of useful options. We describe some of the commonly used options for the setsockopt call. We begin by providing them in the table below. For a more detailed description, we can do "man setsockopt" for the man-pages or google "man setsockopt".

Option Name	Option Level	Description
SO_LINGER	SOL_SOCKET	Send pending data before closing the socket
SO_REUSEADDR	SOL_SOCKET	Allow reuse of a local port that is already bound
SO_KEEPALIVE	SOL_SOCKET	Send periodic keepalives to keep the connection alive
SO_SNDBUF	SOL_SOCKET	Adjust socket send buffer
SO_RECVBUF	SOL_SOCKET	Adjust socket receiver buffer
SO_NONBLOCK	SOL_SOCKET	Makes the socket non-blocking

Table: Some of the setsockopt() options

Now, let us describe these options.

SO_LINGER: This option requests the local socket to send all the pending data present in the send buffer even if the application issues a close. Typically, this is used for TCP where a reliable mode of data transfer is required. The SO_LINGER option usually takes a timeout value which means that after the timeout happens, the close would proceed in a normal manner and if all the present data is not sent within the timeout period, then the remaining would be discarded.

SO_REUSEADDR (level SOL_SOCKET): This option allows a local socket to reuse a local address that may be already in use. In other words, even if there is a socket that is already bound to that local address, a new socket can specify this option and still use the same local address. Binding multiple sockets to the same port and address is often used when we have multiple multicast receivers sitting on the same machine. On certain operating systems, this option might be available as SO_REUSEPORT; this option allows reuse of both the local address and the local port. However, the behavior of these options can be platform specific and should be considered before using it. Further, on some of the platforms (including Linux), both the sockets must set the SO_REUSEADDR, else the second socket not be able to bind.

SO_KEEPALIVE (level SOL_SOCKET): This option allows an application to close a connected socket if the remote end closes without informing the local socket. As the name suggests, when this option is set and if the connection becomes idle (that is the no data to received and sent), then the socket begins to send keepalive messages. If the remote socket does not respond to a pre-specified number of keepalive retries, then the local end closes the connection.

SO_SNDBUF and SO_RECVBUF (level SOL_SOCKET): These options allow an application to control the local socket buffer for send-side and receive-side. If not set, the default value provided by the kernel is used. This buffer optimization can be helpful if the application expects either a high volume of traffic or a low volume of traffic. With high volume traffic, the current default limit might pose a problem and the throughput can decrease due to local buffer limitation. With low volume traffic, the memory consumed by the default limit might go unused; by specifying a lower value, the unused memory can be used somewhere else. Thus, the application can use this option to increase/decrease the local buffers, as per the expected traffic.

SO_NONBLOCK (level SOL_SOCKET): Setting SO_NONBLOCK makes the current socket non-blocking. This would modify the behavior of the usual blocking calls, like recv(), accept(), and connect(). Normally, these calls block if the resource is not available. For example, if there is no received data, then the recv() call would block. But with SO_NONBOCK options, instead of blocking, these calls would return with a value of -1 and errno set to EAGAIN/EWOULDBLOCK. In that case, the application would need to retry later.

All of the options in the above table have an option level of SOL_SOCKET. This means that these options apply in the socket layer of the stack. However, there are other options levels as well: SOL_TCP, SOL_IP, and SOL_IPV6. SOL_TCP is used for options that modify the socket behavior the TCP layer. SOL_IP and SOL_IPV6 are levels that modify the socket behavior for IP and IPv6 layers, respectively.

Next, we present two examples to help us understand the behavior of these two calls.

Our first example sets some of the above options on a socket. After that, we retrieve the earlier set option using the getsockopt() call. For this example, we do not verify the actual behavior corresponding to the options in this example -- that would be beyond the scope of the current text. Also, the SOL_LINGER option accepts the input in the form of a data structure" struct linger. This structure has two fields: (a) int l_onoff and (b) int l_linger. If l_onoff is nonzero, then this option is applied. The l_linger value specifies the timeout period in seconds.

```
 #include <stdio.h>
 #include <errno.h>
 #include <sys/socket.h>
 #include <netinet/in.h>

 int main () {
     struct sockaddr_in saddr;
     int fd, ret_val, opt_len, opt_val = 1, sockbuf_val = 4096;
     struct linger set_linger, get_linger;

     /* First step is to open a socket  */
     fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
     if (fd == -1) {
         fprintf(stderr, "socket call failed [%s]\n", strerror(errno));
         return -1;
     }

     /* Next, let us use setsockopt() to set some options */
     printf("Let us set some socket options\n");
     ret_val = setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &opt_val,
sizeof(int));
     if (ret_val != 0) {
         fprintf(stderr, "setsockopt [SO_KEEPALIVE] failed [%s]\n",
                 strerror(errno));
     } else {
         printf("setsockopt [SO_KEEPALIVE] succeeded for socket fd: %d\n",
fd);
     }

     set_linger.l_onoff = 1;
     set_linger.l_linger = 10;
     ret_val = setsockopt(fd, SOL_SOCKET, SO_LINGER, &set_linger,
             sizeof(struct linger));
     if (ret_val != 0) {
         fprintf(stderr, "setsockopt [SO_LINGER] failed [%s]\n",
                 strerror(errno));
     } else {
         printf("setsockopt [SO_LINGER] succeeded for socket fd: %d\n", fd);
     }

     ret_val = setsockopt(fd, SOL_SOCKET, SO_RCVBUF, &sockbuf_val,
sizeof(int));
     if (ret_val != 0) {
         fprintf(stderr, "setsockopt [SO_RCVBUF] failed [%s]\n",
                 strerror(errno));
     } else {
         printf("setsockopt [SO_RCVBUF] succeeded for socket fd: %d\n", fd);
     }

     /* After that, let us use getsockopt() to retrieve values of options */
     printf("\nLet us retrieve those options:\n");
     opt_len = sizeof(int);
     ret_val = getsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &opt_val, &opt_len);
     if (ret_val != 0) {
         fprintf(stderr, "getsockopt [SO_KEEPALIVE] failed [%s]\n",
                 strerror(errno));
     } else {
```

```
        printf("getsockopt [SO_KEEPALIVE]: value: %d len: %d\n",
            opt_val, opt_len);
    }

    opt_len = sizeof(struct linger);
    ret_val = getsockopt(fd, SOL_SOCKET, SO_LINGER, &get_linger, &opt_len);
    if (ret_val != 0) {
        fprintf(stderr, "getsockopt [SO_LINGER] failed [%s]\n",
                strerror(errno));
    } else {
        printf("getsockopt [SO_LINGER]: l_onoff: %d l_linger: %d len: %d\n",
            get_linger.l_onoff, get_linger.l_linger, opt_len);
    }

    opt_len = sizeof(int);
    ret_val = getsockopt(fd, SOL_SOCKET, SO_RCVBUF, &opt_val, &opt_len);
    if (ret_val != 0) {
        fprintf(stderr, "getsockopt [SO_RCVBUF] failed [%s]\n",
                strerror(errno));
    } else {
        printf("getsockopt [SO_RCVBUF]: value: %d len: %d\n",
            opt_val, opt_len);
    }

    /* Last step is to close the sockets  */
    close(fd);
    return 0;
}
$ gcc setsockopt-socket1.c -o setsockopt1
$
$ ./setsockopt1
Let us set some socket options:
setsockopt [SO_KEEPALIVE] succeeded for socket fd: 3
setsockopt [SO_LINGER] succeeded for socket fd: 3
setsockopt [SO_RCVBUF] succeeded for socket fd: 3

Let us retrieve those options:
getsockopt [SO_KEEPALIVE]: value: 1 len: 4
getsockopt [SO_LINGER]: l_onoff: 1 l_linger: 10 len: 8
getsockopt [SO_RCVBUF]: value: 8192 len: 4
```

The reason why Linux returns double the size of SO_RCVBUF is because Linux actually allocates double the size of what is requested by the user; it does so for storing extra packet buffer needed for each packets.

Our second example sets the SO_REUSEADDR option and also demonstrates its behavior. For that, we set this option on a socket and bind it to an address and a port. After that, we open another socket and try to reuse the same address and port number. Since we have SO_REUSEADDR, we are able to do so! Note that both sockets must set this option, otherwise the second bind() call would fail.

```
#include <stdio.h>
#include <sys/socket.h>
#include <errno.h>
```

```
 #include <netinet/in.h>

 int main () {
     struct sockaddr_in saddr;
     int fd1, fd2;
     int ret_val, opt_val = 1, opt_len = sizeof(int);

     /* First step is to open a socket  */
     fd1 = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
     if (fd1 == -1) {
         fprintf(stderr, "socket call failed [%s]\n", strerror(errno));
         return -1;
     }
     printf("Created a socket with fd: %d\n", fd1);

     /* Next, let us use setsockopt() to set the SO_REUSEADDR option */
     ret_val = setsockopt(fd1, SOL_SOCKET, SO_REUSEADDR, &opt_val,
sizeof(opt_val));
     if (ret_val != 0) {
         fprintf(stderr, "setsockopt call failed [%s]\n", strerror(errno));
     } else {
         printf("setsockopt [SO_REUSEADDR] succeeded for socket fd: %d\n",
fd1);
     }

     /* After that, let us use getsockopt() to retrieve values of options */
     ret_val = getsockopt(fd1, SOL_SOCKET, SO_REUSEADDR, &opt_val, &opt_len);
     if (ret_val != 0) {
         fprintf(stderr, "getsockopt call failed [%s]\n", strerror(errno));
     } else {
         printf("getsockopt [SO_REUSEADDR]: value: %d len: %d\n",
             opt_val, opt_len);
     }

     printf("\nLet us use SO_REUSEADDR to bind two sockets to the same
port\n");
     saddr.sin_family = AF_INET;
     saddr.sin_port = htons(7000);
     saddr.sin_addr.s_addr = INADDR_ANY;

     ret_val = bind(fd1, (struct sockaddr *)&saddr, sizeof(struct sockaddr));
     if (ret_val != 0) {
         fprintf(stderr, "bind call failed [%s]\n", strerror(errno));
     } else {
         printf("bind succeeded for socket fd: %d\n", fd1);
     }

     /* Now, let us create another socket and verify if we can reuse the
address/port */
     fd2 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
     if (fd2 == -1) {
         fprintf(stderr, "socket call failed [%s]\n", strerror(errno));
     }
     printf("Created a socket with fd: %d\n", fd2);

     ret_val = setsockopt(fd2, SOL_SOCKET, SO_REUSEADDR, &opt_val,
sizeof(opt_val));
```

```
    if (ret_val != 0) {
        fprintf(stderr, "setsockopt call failed [%s]\n", strerror(errno));
    } else {
        printf("setsockopt [SO_REUSEADDR] succeeded for socket fd: %d\n",
fd2);
    }

    ret_val = bind(fd2, (struct sockaddr *)&saddr, sizeof(struct sockaddr));
    if (ret_val != 0) {
        fprintf(stderr, "bind call failed [%s]\n", strerror(errno));
    } else {
        printf("bind succeeded for socket fd: %d\n", fd2);
    }

    /* Last step is to close the sockets  */
    close(fd1);
    close(fd2);
    return 0;
}
```

Here is the output:

```
$ gcc setsockopt-socket.c -o setsockopt2
$
$ ./setsockopt2
Created a socket with fd: 3
setsockopt [SO_REUSEADDR] succeeded for socket fd: 3
getsockopt [SO_REUSEADDR]: value: 1 len: 4

Let us use SO_REUSEADDR to bind two sockets to the same port
bind succeeded for socket fd: 3
Created a socket with fd: 4
setsockopt [SO_REUSEADDR] succeeded for socket fd: 4
```

THE END