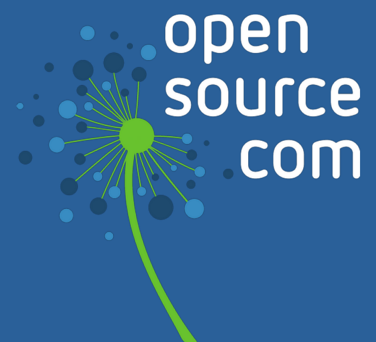



# Tips and tricks for C programming

by Jim Hall

# We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

# Table of Contents

5 common bugs in C programming and how to fix them....4
Write a guessing game in ncurses....13
Position text with ncurses....18
Write a chess game using bit-fields and masks....22
Short option parsing using getopt....26
Learn how file input and output works in C....32
Learn C by writing a simple game....38
Parsing data with strtok in C....41
Programming on FreeDos: Print a Halloween greeting with ASCII art....46
Get started programming with DOS conio....51
How to program in C on FreeDOS....61

The basics of programming with C....67
C programming cheat sheet....74
Appendix: How C has grown....76
The monumental impact of C....80
How to write a good C main function....85

# Jim Hall

Jim Hall is an open source software advocate and developer, best known for usability testing in GNOME and as the founder and project coordinator of FreeDOS.

At work, Jim is CEO of Hallmentum, an IT executive consulting company that provides hands-on IT Leadership training, workshops, and coaching.

# 5 common bugs in C programming and how to fix them

By Jim Hall

Even the best programmers can create programming bugs. Depending on what your program does, these bugs could introduce security vulnerabilities, cause the program to crash, or create unexpected behavior.

The C programming language sometimes gets a bad reputation because it is not memory safe like more recent programming languages, including Rust. But with a little extra code, you can avoid the most common and most serious C programming bugs. Here are five bugs that can break your application and how you can avoid them:

## 1. Uninitialized variables

When the program starts up, the system will assign it a block of memory that the program uses to store data. That means your variables will get whatever random value was in memory when the program started.

Some environments will intentionally "zero out" the memory as the program starts up, so every variable starts with a zero value. And it can be tempting to assume in your programs that all variables will begin at zero. However, the C programming specification says that the system does not initialize variables.

Consider a sample program that uses a few variables and two arrays:

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

main()
{
```

```
  int i, j, k;
  int numbers[5];
  int *array;
  puts("These variables are not initialized:");
  printf("  i = %d\n", i);
  printf("  j = %d\n", j);
  printf("  k = %d\n", k);
  puts("This array is not initialized:");
  for (i = 0; i < 5; i++) {
    printf("  numbers[%d] = %d\n", i, numbers[i]);
  }
  puts("malloc an array ...");
  array = malloc(sizeof(int) * 5);
  if (array) {
    puts("This malloc'ed array is not initialized:");
    for (i = 0; i < 5; i++) {
      printf("  array[%d] = %d\n", i, array[i]);
    }
    free(array);
  }
  /* done */
  puts("Ok");
  return 0;
}
```

The program does not initialize the variables, so they start with whatever values the system had in memory at the time. Compiling and running this program on my Linux system, you'll see that some variables happen to have "zero" values, but others do not:

```
These variables are not initialized:
  i = 0
  j = 0
  k = 32766
This array is not initialized:
  numbers[0] = 0
  numbers[1] = 0
  numbers[2] = 4199024
  numbers[3] = 0
  numbers[4] = 0
malloc an array ...
This malloc'ed array is not initialized:
  array[0] = 0
  array[1] = 0
  array[2] = 0
  array[3] = 0
  array[4] = 0
Ok
```

Fortunately, the `i` and `j` variables start at zero, but `k` has a starting value of 32766. In the numbers array, most elements also happen to start with zero, except the third element, which gets an initial value of 4199024.

Compiling the same program on a different system further shows the danger in uninitialized variables. Don't assume "all the world runs Linux" because one day, your program might run on a different platform. For example, here's the same program running on FreeDOS:

```
These variables are not initialized:
  i = 0
  j = 1074
  k = 3120
This array is not initialized:
  numbers[0] = 3106
  numbers[1] = 1224
  numbers[2] = 784
  numbers[3] = 2926
  numbers[4] = 1224
malloc an array ...
This malloc'ed array is not initialized:
  array[0] = 3136
  array[1] = 3136
  array[2] = 14499
  array[3] = -5886
  array[4] = 219
Ok
```

Always initialize your program's variables. If you assume a variable will start with a zero value, add the extra code to assign zero to the variable. This extra bit of typing upfront will save you headaches and debugging later on.

## 2. Going outside of array bounds

In C, arrays start at array index zero. That means an array that is ten elements long goes from 0 to 9, or an array that is a thousand elements long goes from 0 to 999.

Some programmers sometimes forget this and introduce "off by one" bugs where they reference the array starting at one. In an array that is five elements long, the value the programmer intended to find at array element "5" is not actually the fifth element of the array. Instead, it is some other value in memory, not associated with the array at all.

Here's an example that goes well outside the array bounds. The program starts with an array that's only five elements long but references array elements from outside that range:

```
#include <stdio.h>
#include <stdlib.h>
int
main()
{
  int i;
  int numbers[5];
  int *array;
  /* test 1 */
  puts("This array has five elements (0 to 4)");
  /* initalize the array */
  for (i = 0; i < 5; i++) {
    numbers[i] = i;
  }
  /* oops, this goes beyond the array bounds: */
  for (i = 0; i < 10; i++) {
    printf("  numbers[%d] = %d\n", i, numbers[i]);
  }
  /* test 2 */
  puts("malloc an array ...");
  array = malloc(sizeof(int) * 5);
  if (array) {
    puts("This malloc'ed array also has five elements (0 to 4)");
    /* initalize the array */
    for (i = 0; i < 5; i++) {
      array[i] = i;
    }
    /* oops, this goes beyond the array bounds: */
    for (i = 0; i < 10; i++) {
      printf("  array[%d] = %d\n", i, array[i]);
    }
    free(array);
  }
  /* done */
  puts("Ok");
  return 0;
}
```

Note that the program initializes all the values of the array, from 0 to 4, but then tries to read 0 to 9 instead of 0 to 4. The first five values are correct, but after that you don't know what the values will be:

```
This array has five elements (0 to 4)

  numbers[0] = 0
  numbers[1] = 1
  numbers[2] = 2
```

```
  numbers[3] = 3
  numbers[4] = 4
  numbers[5] = 0
  numbers[6] = 4198512
  numbers[7] = 0
  numbers[8] = 1326609712
  numbers[9] = 32764
malloc an array ...
This malloc'ed array also has five elements (0 to 4)
  array[0] = 0
  array[1] = 1
  array[2] = 2
  array[3] = 3
  array[4] = 4
  array[5] = 0
  array[6] = 133441
  array[7] = 0
  array[8] = 0
  array[9] = 0
Ok
```

When referencing arrays, always keep track of its size. Store that in a variable; don't hard-code an array size. Otherwise, your program might stray outside the array bounds when you later update it to use a different array size, but you forget to change the hard-coded array length.

## 3. Overflowing a string

Strings are just arrays of a different kind. In the C programming language, a string is an array of `char` values, with a zero character to indicate the end of the string.

And so, like arrays, you need to avoid going outside the range of the string. This is sometimes called *overflowing a string*.

One easy way to overflow a string is to read data with the `gets` function. The `gets` function is very dangerous because it doesn't know how much data it can store in a string, and it naively reads data from the user. This is fine if your user enters short strings like `foo` but can be disastrous when the user enters a value that is too long for your string value.

Here's a sample program that reads a city name using the `gets` function. In this program, I've also added a few unused variables to show how string overflow can affect other data:

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

```
main()
{
  char name[10];                       /* Such as "Chicago" */
  int var1 = 1, var2 = 2;
  /* show initial values */
  printf("var1 = %d; var2 = %d\n", var1, var2);
  /* this is bad .. please don't use gets */
  puts("Where do you live?");
  gets(name);
  /* show ending values */
  printf("<%s> is length %d\n", name, strlen(name));
  printf("var1 = %d; var2 = %d\n", var1, var2);
  /* done */
  puts("Ok");
  return 0;
}
```

That program works fine when you test for similarly short city names, like `Chicago` in Illinois or `Raleigh` in North Carolina:

```
var1 = 1; var2 = 2
Where do you live?
Raleigh
<Raleigh> is length 7
var1 = 1; var2 = 2
Ok
```

The Welsh town of `Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch` has one of the longest names in the world. At 58 characters, this string goes well beyond the 10 characters reserved in the `name` variable. As a result, the program stores values in other areas of memory, including the values of `var1` and `var2`:

```
var1 = 1; var2 = 2
Where do you live?
Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch
<Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch> is length 58
var1 = 2036821625; var2 = 2003266668
Ok
Segmentation fault (core dumped)
```

Before aborting, the program used the long string to overwrite other parts of memory. Note that `var1` and `var2` no longer have their starting values of `1` and `2`.

Avoid `gets`, and use safer methods to read user data. For example, the `getline` function will allocate enough memory to store user input, so the user cannot accidentally overflow the string by entering a long value.

## 4. Freeing memory twice

One of the rules of good C programming is, "if you allocate memory, you should free it." Programs can allocate memory for arrays and strings using the `malloc` function, which reserves a block of memory and returns a pointer to the starting address in memory. Later, the program can release the memory using the `free` function, which uses the pointer to mark the memory as unused.

However, you should only use the `free` function once. Calling `free` a second time will result in unexpected behavior that will probably break your program. Here's a short example program to show that. It allocates memory, then immediately releases it. But like a forgetful-but-methodical programmer, I also freed the memory at the end of the program, resulting in freeing the same memory twice:

```
#include <stdio.h>
#include <stdlib.h>
int
main()
{
  int *array;
  puts("malloc an array ...");
  array = malloc(sizeof(int) * 5);
  if (array) {
    puts("malloc succeeded");
    puts("Free the array...");
    free(array);
  }
  puts("Free the array...");
  free(array);
  puts("Ok");
}
```

Running this program causes a dramatic failure on the second use of the `free` function:

```
malloc an array ...
malloc succeeded
Free the array...
Free the array...
free(): double free detected in tcache 2
```

```
Aborted (core dumped)
```

Avoid calling `free` more than once on an array or string. One way to avoid freeing memory twice is to locate the `malloc` and `free` functions in the same function.

For example, a solitaire program might allocate memory for a deck of cards in the main function, then use that deck in other functions to play the game. Free the memory in the main function, rather than some other function. Keeping the `malloc` and `free` statements together helps to avoid freeing memory more than once.

## 5. Using invalid file pointers

Files are a handy way to store data. For example, you might store configuration data for your program in a file called `config.dat`. The Bash shell reads its initial script from `.bash_profile` in the user's home directory. The GNU Emacs editor looks for the file `.emacs` for its starting values. And the Zoom meeting client uses the `zoomus.conf` file to read its program configuration.

So the ability to read data from a file is important for pretty much all programs. But what if the file you want to read isn't there?

To read a file in C, you first open the file using the `fopen` function, which returns a stream pointer to the file. You can use this pointer with other functions to read data, such as `fgetc` to read the file one character at a time.

If the file you want to read isn't there or isn't readable by your program, then the `fopen` function will return `NULL` as the file pointer, which is an indication the file pointer is invalid. But here's a sample program that innocently does not check if `fopen` returned `NULL` and tries to read the file regardless:

```
#include <stdio.h>
int
main()
{
  FILE *pfile;
  int ch;
  puts("Open the FILE.TXT file ...");
  pfile = fopen("FILE.TXT", "r");
  /* you should check if the file pointer is valid, but we skipped that */
  puts("Now display the contents of FILE.TXT ...");
  while ((ch = fgetc(pfile)) != EOF) {
    printf("<%c>", ch);
```

```
  }
  fclose(pfile);
  /* done */
  puts("Ok");
  return 0;
}
```

When you run this program, the first call to `fgetc` results in a spectacular failure, and the program immediately aborts:

```
Open the FILE.TXT file ...
Now display the contents of FILE.TXT ...
Segmentation fault (core dumped)
```

Always check the file pointer to ensure it's valid. For example, after calling `fopen` to open a file, check the pointer's value with something like `if (pfile != NULL)` to ensure that the pointer is something you can use.

We all make mistakes, and programming bugs happen to the best of programmers. But if you follow these guidelines and add a little extra code to check for these five types of bugs, you can avoid the most serious C programming mistakes. A few lines of code up front to catch these errors may save you hours of debugging later.

# Write a guessing game in ncurses

By Jim Hall

In my last article, I gave a brief introduction to using the **ncurses** library to write text-mode interactive applications in C. With **ncurses**, we can control where and how text gets displayed on the terminal. If you explore the **ncurses** library functions by reading the manual pages, you'll find there are a ton of different ways to display text, including bold text, colors, blinking text, windows, borders, graphic characters, and other features to make your application stand out.

If you'd like to explore a more advanced program that demonstrates a few of these interesting features, here's a simple "guess the number" game, updated to use **ncurses**. The program picks a random number in a range, then asks the user to make repeated guesses until they find the secret number. As the user makes their guess, the program lets them know if the guess was too low or too high.

Note that this program limits the possible numbers from 0 to 7. Keeping the values to a limited range of single-digit numbers makes it easier to use **getch()** to read a single number from the user. I also used the **getrandom** kernel system call to generate random bits, masked with the number 7 to pick a random number from 0 (binary 0000) to 7 (binary 0111).

```
#include <curses.h>
#include <string.h>          /* for strlen */
#include <sys/random.h>      /* for getrandom */
int
random0_7()
{
   int num;
   getrandom(&num, sizeof(int), GRND_NONBLOCK);
   return (num & 7); /* from 0000 to 0111 */
}
int
read_guess()
{
```

Creative Commons Attribution Share-alike 4.0 

```
  int ch;
  do {
    ch = getch();
  } while ((ch < '0') || (ch > '7'));
  return (ch - '0'); /* turn into a number */
}
```

By using **ncurses**, we can add some visual interest. Let's add functions to display important text at the top of the screen and a message line to display status information at the bottom of the screen.

```
void
print_header(const char *text)
{
  move(0, 0);
  clrtoeol();
  attron(A_BOLD);
  mvaddstr(0, (COLS / 2) - (strlen(text) / 2), text);
  attroff(A_BOLD);
  refresh();
}
void
print_status(const char *text)
{
  move(LINES - 1, 0);
  clrtoeol();
  attron(A_REVERSE);
  mvaddstr(LINES - 1, 0, text);
  attroff(A_REVERSE);
  refresh();
}
```

With these functions, we can construct the main part of our number-guessing game. First, the program sets up the terminal for **ncurses**, then picks a random number from 0 to 7. After displaying a number scale, the program then enters a loop to ask the user for their guess.

As the user makes their guess, the program provides visual feedback. If the guess is too low, the program prints a left square bracket under the number on the screen. If the guess is too high, the game prints a right square bracket. This helps the user to narrow their choice until they guess the correct number.

```
int
main()
{
  int number, guess;
```

```
  initscr();
  cbreak();
  noecho();
  number = random0_7();
  mvprintw(1, COLS - 1, "%d", number); /* debugging */
  print_header("Guess the number 0-7");
  mvaddstr(9, (COLS / 2) - 7, "0 1 2 3 4 5 6 7");
  print_status("Make a guess...");
  do {
    guess = read_guess();
    move(10, (COLS / 2) - 7 + (guess * 2));
    if (guess < number) {
      addch('[');
      print_status("Too low");
    }
    else if (guess > number) {
      addch(']');
      print_status("Too high");
    }
    else {
      addch('^');
    }
  } while (guess != number);
  print_header("That's right!");
  print_status("Press any key to quit");
  getch();
  endwin();
  return 0;
}
```

Copy this program and compile it for yourself to try it out. Don’t forget that you need to tell GCC to link with the **ncurses** library:

```
$ gcc -o guess guess.c -lncurses
```

I’ve left the debugging line in there, so you can see the secret number near the upper-right corner of the screen:

## Get yourself going with ncurses

This program uses a bunch of other features of **ncurses** that you can use as a starting point. For example, the print_header function prints a message in bold text centered at the top of the screen, and the print_status function prints a message in reverse text at the bottom-left of the screen. Use this to help you get started with **ncurses** programming.

Creative Commons Attribution Share-alike 4.0

# Position text with ncurses

By Jim Hall

Most Linux utilities just scroll text from the bottom of the screen. But what if you wanted to position text on the screen, such as for a game or a data display? That's where **ncurses** comes in.

**curses** is an old Unix library that supports cursor control on a text terminal screen. The name *curses* comes from the term *cursor control*. Years later, others wrote an improved version of **curses** to add new features, called *new curses* or **ncurses**. You can find **ncurses** in every modern Linux distribution, although the development libraries, header files, and documentation may not be installed by default. For example, on Fedora, you will need to install the **ncurses-devel** package with this command:

```
$ sudo dnf install ncurses-devel
```

## Using ncurses in a program

To directly address the screen, you'll first need to initialize the **ncurses** library. Most programs will do that with these three lines:

- initscr(); Initialize the screen and the **ncurses** code
- cbreak(); Disable buffering and make typed input immediately available
- noecho(); Turn off echo, so user input is not displayed to the screen

These functions are defined in the **curses.h** header file, which you'll need to include in your program with:

```
#include <curses.h>
```

After initializing the terminal, you're free to use any of the **ncurses** functions, some of which we'll explore in a sample program.

When you're done with **ncurses** and want to go back to regular terminal mode, use **endwin();** to reset everything. This command resets any screen colors, moves the cursor to the lower-left of the screen, and makes the cursor visible. You usually do this right before exiting the program.

## Addressing the screen

The first thing to know about **ncurses** is that screen coordinates are *row,col*, and start in the upper-left at 0,0. **ncurses** defines two global variables to help you identify the screen size: LINES is the number of lines on the screen, and COLS is the number of columns. The bottom-right position is LINES-1,COLS-1.

For example, if you wanted to move the cursor to line 10 and column 30, you could use the move function with those coordinates:

```
move(10, 30);
```

Any text you display after that will start at that screen location. To display a single character, use the **addch(c)** function with a single character. To display a string, use **addstr(s)** with your string. For formatted output that's similar to **printf**, use **printw(fmt, …)** with the usual options.

Moving to a screen location and displaying text is such a common thing that **ncurses** provides a shortcut to do both at once. The **mvaddch(row, col, c)** function will display a character at screen location *row,col*. And the **mvaddstr(row, col, s)** function will display a string at that location. For a more direct example, using **mvaddstr(10, 30, "Welcome to ncurses");** in a program will display the text "Welcome to ncurses" starting at row 10 and column 30. And the line **mvaddch(0, 0, '+');** will display a single plus sign in the upper-left corner at row 0 and column 0.

Drawing text to the terminal screen can have a performance impact on certain systems, especially on older hardware terminals. So **ncurses** lets you "stack up" a bunch of text to display to the screen, then use the **refresh()** function to make all of those changes visible to the user.

Let's look at a simple example that pulls everything together:

```
#include <curses.h>
int
main()
```

```
{
  initscr();
  cbreak();
  noecho();
  mvaddch(0, 0, '+');
  mvaddch(LINES - 1, 0, '-');
  mvaddstr(10, 30, "press any key to quit");
  refresh();
  getch();
  endwin();
}
```

The program starts by initializing the terminal, then prints a plus sign in the upper-left corner, a minus in the lower-left corner, and the text "press any key to quit" at row 10 and column 30. The program gets a single character from the keyboard using the getch() function, then uses **endwin()** to reset the terminal before the program exits completely.

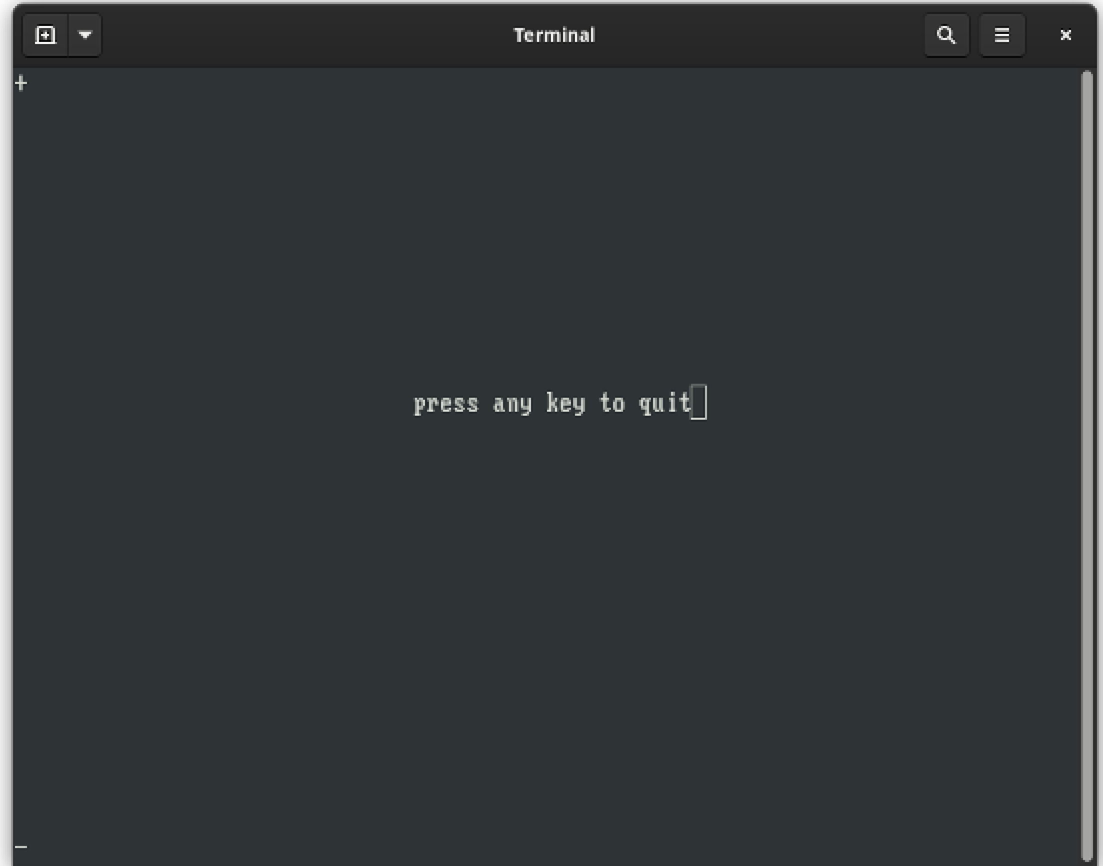
**getch()** is a useful function that you could use for many things. I often use it as a way to pause before I quit the program. And as with most **ncurses** functions, there's also a version of **getch()** called **mvgetch(row, col)** to move to screen position *row,col* before waiting for a character.

## Compiling with ncurses

If you tried to compile that sample program in the usual way, such as `gcc pause.c`, you'll probably get a huge list of errors from the linker. That's because the **ncurses** library is not linked automatically by the GNU C Compiler. Instead, you'll need to load it for linking using the `-l ncurses` command-line option.

```
$ gcc -o pause pause.c -lncurses
```

Running the new program will print a simple "press any key to quit" message that's more or less centered on the screen:




## Building better programs with ncurses

Explore the **ncurses** library functions to learn about other ways to display text to the screen. You can find a list of all **ncurses** functions in the man ncurses manual page. This gives a general overview of **ncurses** and provides a table-like list of the different **ncurses** functions, with a reference to the manual page that has full details. For example, **printw** is described in the *curs_printw(3X)* manual page, which you can view with:

```
$ man 3x curs_printw
```

or just:

```
$ man curs_printw
```

With **ncurses**, you can create more interesting programs. By printing text at specific locations on the screen, you can create games and advanced utilities to run in the terminal.

# Write a chess game using bit-fields and masks

By Jim Hall

Let's say you were writing a chess game in C. One way to track the pieces on the board is by defining a structure that defines each possible piece on the board, and its color, so every square contains an element from that structure. For example, you might have a structure that looks like this:

```
struct chess_pc {
   int piece;
   int is_black;
}
```

With this programming structure, your program will know what piece is in every square and its color. You can quickly identify if the piece is a pawn, rook, knight, bishop, queen, or king—and if the piece is black or white. But there's a more straightforward way to track the same information while using less data and memory. Rather than storing a structure of two `int` values for every square on a chessboard, we can store a single `int` value and use binary *bit-fields* and *masks* to identify the pieces and color in each square.

## Bits and binary

When using bit-fields to represent data, it helps to think like a computer. Let's start by listing the possible chess pieces and assigning a number to each. I'll help us along to the next step by representing the number in its binary form, the way the computer would track it. Remember that binary numbers are made up of *bits*, which are either zero or one.

- **`00000000:`** empty (0)
- **`00000001:`** pawn (1)

Creative Commons Attribution Share-alike 4.0

- **`00000010`**: rook (2)
- **`00000011`**: knight (3)
- **`00000100`**: bishop (4)
- **`00000101`**: queen (5)
- **`00000110`**: king (6)

To list all pieces on a chessboard, we only need the three bits that represent (from right to left) the values 1, 2, and 4. For example, the number 6 is binary `110`. All of the other bits in the binary representation of 6 are zeroes.

And with a bit of cleverness, we can use one of those extra always-zero bits to track if a piece is black or white. We can use the number 8 (binary `00001000`) to indicate if a piece is black. If this bit is 1, it's black; if it's 0, it's white. That's called a *bit-field*, which we can pull out later using a binary *mask*.

## Storing data with bit-fields

To write a chess program using bit-fields and masks, we might start with these definitions:

```
/* game pieces */
#define EMPTY 0
#define PAWN 1
#define ROOK 2
#define KNIGHT 3
#define BISHOP 4
#define QUEEN 5
#define KING 6
/* piece color (bit-field) */
#define BLACK 8
#define WHITE 0
/* piece only (mask) */
#define PIECE 7
```

When you assign a value to a square, such as when initializing the chessboard, you can assign a single `int` value to track both the piece and its color. For example, to store a black rook in position 0,0 of an array, you would use this code:

```
  int board[8][8];
..
  board[0][0] = BLACK | ROOK;
```

The | is a binary OR, which means the computer will combine the bits from two numbers. For every bit position, if that bit from *either* number is 1, the result for that bit position is also 1. Binary OR of the value `BLACK` (8, or binary `00001000`) and the value `ROOK` (2, or binary `00000010`) is binary `00001010`, or 10:

```
   00001000 = 8
OR 00000010 = 2
   ________
   00001010 = 10
```

Similarly, to store a white pawn in position 6,0 of the array, you could use this:

```
  board[6][0] = WHITE | PAWN;
```

This stores the value 1 because the binary OR of `WHITE` (0) and `PAWN` (1) is just 1:

```
   00000000 = 0
OR 00000001 = 1
   ________
   00000001 = 1
```

## Getting data out with masks

During the chess game, the program will need to know what piece is in a square and its color. We can separate the piece using a binary mask.

For example, the program might need to know the contents of a specific square on the board during the chess game, such as the array element at `board[5][3]`. What piece is there, and is it black or white? To identify the chess piece, combine the element's value with the `PIECE` mask using the binary AND:

```
  int board[8][8];
  int piece;
..
  piece = board[5][3] & PIECE;
```

The binary AND operator (&) combines two binary values so that for any bit position, if that bit in *both* numbers is 1, then the result is also 1. For example, if the value of `board[5][3]` is 11 (binary `00001011`), then the binary AND of 11 and the mask PIECE (7, or binary `00000111`) is binary `00000011`, or 3. This is a knight, which also has the value 3.

```
    00001011 = 11
AND 00000111 = 7
    ________
    00000011 = 3
```

Separating the piece's color is a simple matter of using binary AND with the value and the `BLACK` bit-field. For example, you might write this as a function called `is_black` to determine if a piece is either black or white:

```
int
is_black(int piece)
{
  return (piece & BLACK);
}
```

This works because the value `BLACK` is 8, or binary `00001000`. And in the C programming language, any non-zero value is treated as True, and zero is always False. So `is_black(board[5][3])` will return a True value (8) if the piece in array element `5,3` is black and will return a False value (0) if it is white.

## Bit fields

Using bit-fields and masks is a common method to combine data without using structures. They are worth adding to your programmer's "tool kit." While data structures are a valuable tool for ordered programming where you need to track related data, using separate elements to track single On or Off values (such as the colors of chess pieces) is less efficient. In these cases, consider using bit-fields and masks to combine your data more efficiently.

# Short option parsing using getopt

By Jim Hall

Writing a C program to process files is easy when you already know what files you'll operate on and what actions to take. If you "hard code" the filename into your program, or if your program is coded to do things only one way, then your program will always know what to do.

But you can make your program much more flexible if it can respond to the user every time the program runs. Let your user tell your program what files to use or how to do things differently. And for that, you need to read the command line.

## Reading the command line

When you write a program in C, you might start with the declaration:

```
int main()
```

That's the simplest way to start a C program. But if you add these standard parameters in the parentheses, your program can read the options given to it on the command line:

```
int main(int argc, char **argv)
```

The `argc` variable is the argument count or the number of arguments on the command line. This will always be a number that's at least one.

The `argv` variable is a double pointer, an array of strings, that contains the arguments from the command line. The first entry in the array, `*argv[0]`, is always the name of the program. The other elements of the `**argv` array contain the rest of the command-line arguments.

I'll write a simple program to echo back the options given to it on the command line. This is similar to the Linux `echo` command, except it also prints the name of the program. It also prints each command-line option on its own line using the `puts` function:

```
#include <stdio.h>
int
main(int argc, char **argv)
{
  int i;
  printf("argc=%d\n", argc); /* debugging */
  for (i = 0; i < argc; i++) {
    puts(argv[i]);
  }
  return 0;
}
```

Compile this program and run it with some command-line options, and you'll see your command line printed back to you, each item on its own line:

```
$ ./echo this program can read the command line
argc=8
./echo
this
program
can
read
the
command
line
```

This command line sets the program's `argc` to `8`, and the `**argv` array contains eight entries: the name of the program, plus the seven words the user entered. And as always in C programs, the array starts at zero, so the elements are numbered 0, 1, 2, 3, 4, 5, 6, 7. That's why you can process the command line with the `for` loop using the comparison `i < argc`.

You can use this to write your own versions of the Linux `cat` or `cp` commands. The `cat` command's basic functionality displays the contents of one or more files. Here's a simple version of `cat` that reads the filenames from the command line:

```
#include <stdio.h>
void
copyfile(FILE *in, FILE *out)
{
  int ch;
  while ((ch = fgetc(in)) != EOF) {
    fputc(ch, out);
  }
}
int
```

```
main(int argc, char **argv)
{
  int i;
  FILE *fileptr;
  for (i = 1; i < argc; i++) {
    fileptr = fopen(argv[i], "r");
    if (fileptr != NULL) {
      copyfile(fileptr, stdout);
      fclose(fileptr);
    }
  }
  return 0;
}
```

This simple version of `cat` reads a list of filenames from the command line and displays the contents of each file to the standard output, one character at a time. For example, if I have one file called `hello.txt` that contains a few lines of text, I can display its contents with my own `cat` command:

```
$ ./cat hello.txt
Hi there!
This is a sample text file.
```

Using this sample program as a starting point, you can write your own versions of other Linux commands, such as the `cp` program, by reading only two filenames: one file to read from and another file to write the copy.

## Reading command-line options

Reading filenames and other text from the command line is great, but what if you want your program to change its behavior based on the *options* the user gives it? For example, the Linux `cat` command supports several command-line options, including:

- `-b` Put line numbers next to non-blank lines
- `-E` Show the ends of lines as `$`
- `-n` Put line numbers next to all lines
- `-s` Suppress printing repeated blank lines
- `-T` Show tabs as `^I`
- `-v` Verbose; show non-printing characters using `^x` and `M-x` notation, except for new lines and tabs

These *single-letter* options are called *short options*, and they always start with a single hyphen character. You usually see these short options written separately, such as `cat -E -n`, but you can also combine the short options into a single *option string* such as `cat -En`.

Fortunately, there's an easy way to read these from the command line. All Linux and Unix systems include a special C library called `getopt`, defined in the `unistd.h` header file. You can use `getopt` in your program to read these short options.

Unlike other Unix systems, `getopt` on Linux will always ensure your short options appear at the front of your command line. For example, say a user types `cat -E file -n`. The `-E` option is upfront, but the `-n` option is after the filename. But if you use Linux `getopt`, your program will always behave as though the user types `cat -E -n file`. That makes processing a breeze because `getopt` can parse the short options, leaving you a list of filenames on the command line that your program can read using the `**argv` array.

You use `getopt` like this:

```
       #include <unistd.h>
       int getopt(int argc, char **argv, char *optstring);
```

The option string `optstring` contains a list of the valid option characters. If your program only allows the `-E` and `-n` options, you use `"En"` as your option string.

You usually use `getopt` in a loop to parse the command line for options. At each `getopt` call, the function returns the next short option it finds on the command line or the value `'?'` for any unrecognized short options. When `getopt` can't find any more short options, it returns `-1` and sets the global variable `optind` to the next element in `**argv` after all the short options.

Let's look at a simple example. This demo program isn't a full replacement of `cat` with all the options, but it can parse its command line. Every time it finds a valid command-line option, it prints a short message to indicate it was found. In your own programs, you might instead set a variable or take some other action that responds to that command-line option:

```
#include <stdio.h>
#include <unistd.h>
int
main(int argc, char **argv)
{
  int i;
  int option;
```

```
  /* parse short options */
  while ((option = getopt(argc, argv, "bEnsTv")) != -1) {
    switch (option) {
    case 'b':
      puts("Put line numbers next to non-blank lines");
      break;
    case 'E':
      puts("Show the ends of lines as $");
      break;
    case 'n':
      puts("Put line numbers next to all lines");
      break;
    case 's':
      puts("Suppress printing repeated blank lines");
      break;
    case 'T':
      puts("Show tabs as ^I");
      break;
    case 'v':
      puts("Verbose");
      break;
    default:                          /* '?' */
      puts("What's that??");
    }
  }
  /* print the rest of the command line */
  puts("------------------------------");
  for (i = optind; i < argc; i++) {
    puts(argv[i]);
  }
  return 0;
}
```

If you compile this program as `args`, you can try out different command lines to see how they parse the short options and always leave you with the rest of the command line. In the simplest case, with all the options up front, you get this:

```
$ ./args -b -T file1 file2
Put line numbers next to non-blank lines
Show tabs as ^I
------------------------------
file1
file2
```

Now try the same command line but combine the two short options into a single option string:

```
$ ./args -bT file1 file2
Put line numbers next to non-blank lines
Show tabs as ^I
------------------------------
file1
file2
```

If necessary, `getopt` can "reorder" the command line to deal with short options that are out of order:

```
$ ./args -E file1 file2 -T
Show the ends of lines as $
Show tabs as ^I
------------------------------
file1
file2
```

If your user gives an incorrect short option, `getopt` prints a message:

```
$ ./args -s -an file1 file2
Suppress printing repeated blank lines
./args: invalid option -- 'a'
What's that??
Put line numbers next to all lines
------------------------------
file1
file2
```

## Download the cheat sheet

`getopt` can do lots more than what I've shown. For example, short options can take their own options, such as `-s string` or `-f file`. You can also tell `getopt` to not display error messages when it finds unrecognized options. Read the `getopt(3)` manual page using `man 3 getopt` to learn more about what `getopt` can do for you.

If you're looking for gentle reminders on the syntax and structure of `getopt()` and `getopt_long()`, download my getopt cheat sheet. One page demonstrates short options, and the other side demonstrates long options with minimum viable code and a listing of the global variables you need to know.

# Learn how file input and output works in C

By Jim Hall

If you want to learn input and output in C, start by looking at the `stdio.h` include file. As you might guess from the name, that file defines all the standard ("std") input and output ("io") functions.

The first `stdio.h` function that most people learn is the `printf` function to print formatted output. Or the `puts` function to print a simple string. Those are great functions to print information to the user, but if you want to do more than that, you'll need to explore other functions.

You can learn about some of these functions and methods by writing a replica of a common Linux command. The `cp` command will copy one file to another. If you look at the `cp` man page, you'll see that `cp` supports a broad set of command-line parameters and options. But in the simplest case, `cp` supports copying one file to another:

```
cp infile outfile
```

You can write your own version of this `cp` command in C by using only a few basic functions to *read* and *write* files.

## Reading and writing one character at a time

You can easily do input and output using the `fgetc` and `fputc` functions. These read and write data one character at a time. The usage is defined in `stdio.h` and is quite straightforward: `fgetc` reads (gets) a single character from a file, and `fputc` puts a single character into a file.

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

Writing the `cp` command requires accessing files. In C, you open a file using the `fopen` function, which takes two arguments: the *name* of the file and the *mode* you want to use. The mode is usually `r` to read from a file or `w` to write to a file. The mode supports other options too, but for this tutorial, just focus on reading and writing.

Copying one file to another then becomes a matter of opening the source and destination files, then *reading one character at a time* from the first file, then *writing that character* to the second file. The `fgetc` function returns either the single character read from the input file or the *end of file* (`EOF`) marker when the file is done. Once you've read `EOF`, you've finished copying and you can close both files. That code looks like this:

```
  do {
    ch = fgetc(infile);
    if (ch != EOF) {
      fputc(ch, outfile);
    }
  } while (ch != EOF);
```

You can write your own `cp` program with this loop to read and write one character at a time by using the `fgetc` and `fputc` functions. The `cp.c` source code looks like this:

```
#include <stdio.h>
int
main(int argc, char **argv)
{
  FILE *infile;
  FILE *outfile;
  int ch;
  /* parse the command line */
  /* usage: cp infile outfile */
  if (argc != 3) {
    fprintf(stderr, "Incorrect usage\n");
    fprintf(stderr, "Usage: cp infile outfile\n");
    return 1;
  }
  /* open the input file */
  infile = fopen(argv[1], "r");
  if (infile == NULL) {
    fprintf(stderr, "Cannot open file for reading: %s\n", argv[1]);
    return 2;
  }
```

```
  /* open the output file */
  outfile = fopen(argv[2], "w");
  if (outfile == NULL) {
    fprintf(stderr, "Cannot open file for writing: %s\n", argv[2]);
    fclose(infile);
    return 3;
  }
  /* copy one file to the other */
  /* use fgetc and fputc */
  do {
    ch = fgetc(infile);
    if (ch != EOF) {
      fputc(ch, outfile);
    }
  } while (ch != EOF);
  /* done */
  fclose(infile);
  fclose(outfile);
  return 0;
}
```

And you can compile that `cp.c` file into a full executable using the GNU Compiler Collection (GCC):

```
$ gcc -Wall -o cp cp.c
```

The `-o cp` option tells the compiler to save the compiled program into the `cp` program file. The `-Wall` option tells the compiler to turn on all warnings. If you don't see any warnings, that means everything worked correctly.

## Reading and writing blocks of data

Programming your own `cp` command by reading and writing data one character at a time does the job, but it's not very fast. You might not notice when copying "everyday" files like documents and text files, but you'll really notice the difference when copying large files or when copying files over a network. Working on one character at a time requires significant overhead.

A better way to write this `cp` command is by reading a chunk of the input into memory (called a *buffer*), then writing that collection of data to the second file. This is much faster because the program can read more of the data at one time, which requires fewer "reads" from the file.

Creative Commons Attribution Share-alike 4.0

You can read a file into a variable by using the `fread` function. This function takes several arguments: the array or memory buffer to read data into (`ptr`), the size of the smallest thing you want to read (`size`), how many of those things you want to read (`nmemb`), and the file to read from (`stream`):

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The different options provide quite a bit of flexibility for more advanced file input and output, such as reading and writing files with a certain data structure. But in the simple case of *reading data from one file* and *writing data to another file*, you can use a buffer that is an array of characters.

And you can write the buffer to another file using the `fwrite` function. This uses a similar set of options to the `fread` function: the array or memory buffer to read data from, the size of the smallest thing you need to write, how many of those things you need to write, and the file to write to.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

In the case where the program reads a file into a buffer, then writes that buffer to another file, the array (`ptr`) can be an array of a fixed size. For example, you can use a `char` array called `buffer` that is 200 characters long.

With that assumption, you need to change the loop in your `cp` program to *read data from a file into a buffer* then *write that buffer to another file*:

```
  while (!feof(infile)) {
    buffer_length = fread(buffer, sizeof(char), 200, infile);
    fwrite(buffer, sizeof(char), buffer_length, outfile);
  }
```

Here's the full source code to your updated `cp` program, which now uses a buffer to read and write data:

```
#include <stdio.h>
int
main(int argc, char **argv)
{
  FILE *infile;
  FILE *outfile;
  char buffer[200];
  size_t buffer_length;
```

```
  /* parse the command line */
  /* usage: cp infile outfile */
  if (argc != 3) {
    fprintf(stderr, "Incorrect usage\n");
    fprintf(stderr, "Usage: cp infile outfile\n");
    return 1;
  }
  /* open the input file */
  infile = fopen(argv[1], "r");
  if (infile == NULL) {
    fprintf(stderr, "Cannot open file for reading: %s\n", argv[1]);
    return 2;
  }
  /* open the output file */
  outfile = fopen(argv[2], "w");
  if (outfile == NULL) {
    fprintf(stderr, "Cannot open file for writing: %s\n", argv[2]);
    fclose(infile);
    return 3;
  }
  /* copy one file to the other */
  /* use fread and fwrite */
  while (!feof(infile)) {
    buffer_length = fread(buffer, sizeof(char), 200, infile);
    fwrite(buffer, sizeof(char), buffer_length, outfile);
  }
  /* done */
  fclose(infile);
  fclose(outfile);
  return 0;
}
```

Since you want to compare this program to the other program, save this source code as `cp2.c`. You can compile that updated program using GCC:

```
$ gcc -Wall -o cp2 cp2.c
```

As before, the `-o cp2` option tells the compiler to save the compiled program into the `cp2` program file. The `-Wall` option tells the compiler to turn on all warnings. If you don't see any warnings, that means everything worked correctly.

## Yes, it really is faster

Reading and writing data using buffers is the better way to write this version of the `cp` program. Because it reads chunks of a file into memory at once, the program doesn't need to

read data as often. You might not notice a difference in using either method on smaller files, but you'll really see the difference if you need to copy something that's much larger or when copying data on slower media like over a network connection.

I ran a runtime comparison using the Linux `time` command. This command runs another program, then tells you how long that program took to complete. For my test, I wanted to see the difference in time, so I copied a 628MB CD-ROM image file I had on my system.

I first copied the image file using the standard Linux `cp` command to see how long that takes. By running the Linux `cp` command first, I also eliminated the possibility that Linux's built-in file-cache system wouldn't give my program a false performance boost. The test with Linux `cp` took much less than one second to run:

```
$ time cp FD13LIVE.iso tmpfile
real    0m0.040s
user    0m0.001s
sys     0m0.003s
```

Copying the same file using my own version of the `cp` command took significantly longer. Reading and writing one character at a time took almost five seconds to copy the file:

```
$ time ./cp FD13LIVE.iso tmpfile
real    0m4.823s
user    0m4.100s
sys     0m0.571s
```

Reading data from an input into a buffer and then writing that buffer to an output file is much faster. Copying the file using this method took less than a second:

```
$ time ./cp2 FD13LIVE.iso tmpfile
real    0m0.944s
user    0m0.224s
sys     0m0.608s
```

My demonstration `cp` program used a buffer that was 200 characters. I'm sure the program would run much faster if I read more of the file into memory at once. But for this comparison, you can already see the huge difference in performance, even with a small, 200 character buffer.

# Learn C by writing a simple game

By Jim Hall

I taught myself about programming back in elementary school. My first programs were on the Apple II, but eventually, I learned C by reading books and practicing. And the best way to practice programming is to write sample programs that help exercise your new knowledge.

One program I like to write in a new language is a simple "guess the number" game. The computer picks a random number from 1 to 100, and you have to figure it out by making guesses. In another article, I showed how to write this "Guess the number" game in Bash, and my fellow Opensource.com authors have written articles about how to write it in Java, Julia, and other computer languages.

What's great about a "Guess the number" game is that it exercises several programming concepts: how to use variables, how to compare values, how to print output, and how to read input.

Over the summer, I recorded a video series to teach people how to write programs in the C programming language. Since then, I've heard from many people who are learning C programming by following it. So, I thought I'd follow up by writing a "Guess the number" game in C.

## Pick a random number

Start the "Guess the number" game by writing a function to pick a random number. When writing functions, good programmers try to make them flexible, so they can reuse them to solve slightly different problems. So, instead of hard-coding the function to pick a random number between 1 and 100, write the function to pick a random number between 1 and some integer value `maxval`:

```
#include <stdio.h>
#include <sys/random.h>
```

```
int
randnum(int maxval)
{
  /* pick a random number from 1 to maxval */
  int randval;
  getrandom(&randval, sizeof(int), GRND_NONBLOCK);
  /* could be negative, so ensure it's positive */
  if (randval < 0) {
    return (-1 * randval % maxval + 1);
  }
  else {
    return (randval % maxval + 1);
  }
}
```

The function uses the Linux system call `getrandom` to generate a series of random bits. You can learn more about this system call on the man page, but note that `getrandom` will fill the variable with random zeroes and ones. That means the final value could be positive or negative, so you need to do a test afterward to ensure the result of your `randnum` function is a positive value.

## Write the program

You can use this function to write your "Guess the number" program:

```
#include <stdio.h>
#include <sys/random.h>
int
randnum(int maxval)
{
  ...
}
int
main(void)
{
  int number;
  int guess;
  number = randnum(100);
  puts("Guess a number between 1 and 100");
  do {
    scanf("%d", &guess);
    if (guess < number) {
      puts("Too low");
    }
    else if (guess > number) {
```

```
      puts("Too high");
    }
  } while (guess != number);
  puts("That's right!");
  return 0;
}
```

The program starts by picking a random number between 1 and 100 using the `randnum` function. After printing a prompt to the user, the program enters a `do-while` loop so the user can guess the number.

In each iteration of the loop, the program tests the user's guess. If the user's guess is less than the random number, the program prints "Too low," and if the guess is greater than the random number, the program prints "Too high." The loop continues until the user's guess is the same as the random number.

When the loop exits, the program prints "That's right!" and then immediately ends.

```
$ gcc -o guess -Wall guess.c
$ ./guess
Guess a number between 1 and 100
50
Too high
30
Too low
40
Too low
45
Too high
42
Too low
43
Too low
44
That's right!
```

## Try it out

This "guess the number" game is a great introductory program when learning a new programming language because it exercises several common programming concepts in a pretty straightforward way. By implementing this simple game in different programming languages, you can demonstrate some core concepts and compare details in each language.

# Parsing data with strtok in C

By Jim Hall

Some programs can just process an entire file at once, and other programs need to examine the file line-by-line. In the latter case, you likely need to parse data in each line. Fortunately, the C programming language has a standard C library function to do just that.

The **strtok** function breaks up a line of data according to "delimiters" that divide each field. It provides a streamlined way to parse data from an input string.

## Reading the first token

Suppose your program needs to read a data file, where each line is separated into different fields with a semicolon. For example, one line from the data file might look like this:

```
102*103;K1.2;K0.5
```

In this example, store that in a string variable. You might have read this string into memory using any number of methods. Here's the line of code:

```
char string[] = "102*103;K1.2;K0.5";
```

Once you have the line in a string, you can use **strtok** to pull out "tokens." Each token is part of the string, up to the next delimiter. The basic call to **strtok** looks like this:

```
#include <string.h>
char *strtok(char *string, const char *delim);
```

The first call to **strtok** reads the string, adds a null (`\0`) character at the first delimiter, then returns a pointer to the first token. If the string is already empty, **strtok** returns NULL.

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

Creative Commons Attribution Share-alike 4.0 

```
int
main()
{
  char string[] = "102*103;K1.2;K0.5";
  char *token;
  token = strtok(string, ";");
  if (token == NULL) {
    puts("empty string!");
    return 1;
  }
  puts(token);
  return 0;
}
```

This sample program pulls off the first token in the string, prints it, and exits. If you compile this program and run it, you should see this output:

```
102*103
```

102*103 is the first part of the input string, up to the first semicolon. That's the first token in the string.

Note that calling **strtok** modifies the string you are examining. If you want the original string preserved, make a copy before using **strtok**.

## Reading the rest of the string as tokens

Separating the rest of the string into tokens requires calling **strtok** multiple times until all tokens are read. After parsing the first token with **strtok**, any further calls to **strtok** must use NULL in place of the string variable. The NULL allows **strtok** to use an internal pointer to the next position in the string.

Modify the sample program to read the rest of the string as tokens. Use a while loop to call **strtok** multiple times until you get NULL.

```
#include <stdio.h>
#include <string.h>
int
main()
{
  char string[] = "102*103;K1.2;K0.5";
  char *token;
  token = strtok(string, ";");
```

```
  if (token == NULL) {
    puts("empty string!");
    return 1;
  }
  while (token) {
    /* print the token */
    puts(token);
    /* parse the same string again */
    token = strtok(NULL, ";");
  }
  return 0;
}
```

By adding the while loop, you can parse the rest of the string, one token at a time. If you compile and run this sample program, you should see each token printed on a separate line, like this:

```
102*103
K1.2
K0.5
```

## Multiple delimiters in the input string

Using **strtok** provides a quick and easy way to break up a string into just the parts you're looking for. You can use **strtok** to parse all kinds of data, from plain text files to complex data. However, be careful that multiple delimiters next to each other are the same as one delimiter.

For example, if you were reading CSV data (comma-separated values, such as data from a spreadsheet), you might expect a list of four numbers to look like this:

```
1,2,3,4
```

But if the third "column" in the data was empty, the CSV might instead look like this:

```
1,2,,4
```

This is where you need to be careful with **strtok**. With **strtok**, multiple delimiters next to each other are the same as a single delimiter. You can see this by modifying the sample program to call **strtok** with a comma delimiter:

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

```
int
main()
{
  char string[] = "1,2,,4";
  char *token;
  token = strtok(string, ",");
  if (token == NULL) {
    puts("empty string!");
    return 1;
  }
  while (token) {
    puts(token);
    token = strtok(NULL, ",");
  }
  return 0;
}
```

If you compile and run this new program, you'll see **strtok** interprets the `,,` as a single comma and parses the data as three numbers:

```
1
2
4
```

Knowing this limitation in **strtok** can save you hours of debugging.

## Using multiple delimiters in strtok

You might wonder why the **strtok** function uses a string for the delimiter instead of a single character. That's because **strtok** can look for different delimiters in the string. For example, a string of text might have spaces and tabs between each word. In this case, you would use each of those "whitespace" characters as delimiters:

```
#include <stdio.h>
#include <string.h>
int
main()
{
  char string[] = "  hello \t world";
  char *token;
  token = strtok(string, " \t");
  if (token == NULL) {
    puts("empty string");
    return 1;
  }
```

```
  while (token) {
    puts(token);
    token = strtok(NULL, " \t");
  }
  return 0;
}
```

Each call to **strtok** uses both a space and tab character as the delimiter string, allowing **strtok** to parse the line correctly into two tokens.

## Wrap up

The **strtok** function is a handy way to read and interpret data from strings. Use it in your next project to simplify how you read data into your program.

# Programming on FreeDos: Print a Halloween greeting with ASCII art

By Jim Hall

Full-color ASCII art used to be quite popular on DOS, which could leverage the extended ASCII character set and its collection of drawing elements. You can add a little visual interest to your next FreeDOS program by adding ASCII art as a cool "welcome" screen or as a colorful "exit" screen with more information about the program.

But this style of ASCII art isn't limited just to FreeDOS applications. You can use the same method in a Linux terminal-mode program. While Linux uses ncurses to control the screen instead of DOS's conio, the related concepts apply well to Linux programs. This article looks at how to generate colorful ASCII art from a C program.

## An ASCII art file

You can use a variety of tools to draw your ASCII art. For this example, I used an old DOS application called TheDraw, but you can find modern open source ASCII art programs on Linux, such as Moebius (Apache license) or PabloDraw (MIT license). It doesn't matter what tool you use as long as you know what the saved data looks like.

Here's part of a sample ASCII art file, saved as C source code. Note that the code snippet defines a few values: `IMAGEDATA_WIDTH` and `IMAGEDATA_DEPTH` define the number of columns and rows on the screen. In this case, it's an 80x25 ASCII art "image." `IMAGEDATA_LENGTH` defines the number of entries in the `IMAGEDATA` array. Each character in the ASCII art screen can be represented by two bytes of data: The character to display and a color attribute containing both the foreground and background colors for the character. For an 80x25 screen, where each character is paired with an attribute, the array contains 4000 entries (that's 80 * 25 * 2 = 4000).

Creative Commons Attribution Share-alike 4.0 

```
#define IMAGEDATA_WIDTH 80
#define IMAGEDATA_DEPTH 25
#define IMAGEDATA_LENGTH 4000
unsigned char IMAGEDATA [] = {
    '.', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,
    ' ', 0x08,  ' ', 0x08,  '.', 0x0F,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,
    ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  '.', 0x0F,
    ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,
    ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,  ' ', 0x08,
```

and so on for the rest of the array.

To display this ASCII art to the screen, you need to write a small program to read the array and print each character with the right colors.

## Setting a color attribute

The color attribute in this ASCII art file defines both the background and foreground color in a single byte, represented by hexadecimal values like `0x08` or `0x6E`. Hexadecimal turns out to be a compact way to express a color "pair" like this.

Character mode systems like ncurses on Linux or conio on DOS can display only sixteen colors. That's sixteen possible text colors and eight background colors. Counting sixteen values (from 0 to 15) in binary requires only four bits:

- `1111` is 16 in binary

And conveniently, hexadecimal can represent 0 to 15 with a single character: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. So the value `F` in hexadecimal is the number 15, or `1111` in binary.

With color pairs, you can encode both the background and foreground colors in a single byte of eight bits. That's four bits for the text color (0 to 15 or 0 to F in hexadecimal) and three bits for the background color (0 to 7 or 0 to E in hexadecimal). The leftover bit in the byte is not used here, so we can ignore it.

To convert the color pair or attribute into color values that your program can use, you'll need to use a bit mask to specify only the bits used for the text color or background color. Using the OpenWatcom C Compiler on FreeDOS, you can write this function to set the colors appropriately from the color attribute:

```
void
textattr(int newattr)
{
```

```
  _settextcolor(newattr & 15);         /* 0000xxxx */
  _setbkcolor((newattr >> 4) & 7);     /* 0xxx0000 */
}
```

The `_settextcolor` function sets just the text color, and the `_setbkcolor` function sets the background color. Both are defined in `graph.h`. Note that because the color attribute included both the background color and the foreground color in a single byte value, the `textattr` function uses & (binary AND) to set a bit mask that isolates only the last four bits in the attribute. That's where the color pair stores the values 0 to 15 for the foreground color.

To get the background color, the function first performs a bit shift to "push" the bits to the right. This puts the "upper" bits into the "lower" bit range, so any bits like `0xxx0000` become `00000xxx` instead. We can use another bit mask with 7 (binary `0111`) to pick out the background color value.

## Displaying ASCII art

The `IMAGEDATA` array contains the entire ASCII art screen and the color values for each character. To display the ASCII art to the screen, your program needs to scan the array, set the color attribute, then show the screen one character at a time.

Let's leave room at the bottom of the screen for a separate message or prompt to the user. That means instead of displaying all 25 lines of an 80-column ASCII screen, I only want to show the first 24 lines.

```
  /* print one line less than the 80x25 that's in there:
     80 x 24 x 2 = 3840 */
  for (pos = 0; pos < 3840; pos += 2) {
...
  }
```

Inside the `for` loop, we need to set the colors, then print the character. The OpenWatcom C Compiler provides a function `_outtext` to display text with the current color values. However, this requires passing a string and would be inefficient if we need to process each character one at a time, in case each character on a line requires a different color.

Instead, OpenWatcom has a similar function called `_outmem` that allows you to indicate how many characters to display. For one character at a time, we can provide a pointer to a character value in the `IMAGEDATA` array and tell `_outtext` to show just one character. That will display the character using the current color attributes, which is what we need.

```
  for (pos = 0; pos < 3840; pos += 2) {
    ch = &IMAGEDATA[pos];              /* pointer assignment */
    attr = IMAGEDATA[pos + 1];

    textattr(attr);
    _outmem(ch, 1);
  }
```

This updated `for` loop sets the character `ch` by assigning a pointer into the `IMAGEDATA` array. Next, the loop sets the text attributes, and then displays the character with `_outmem`.

## Putting it all together

With the `textattr` function and the `for` loop to process the array, we can write a full program to display the contents of an ASCII art file. For this example, save the ASCII art as `imgdata.inc` and include it in the source file with an `#include` statement.

```
#include <stdio.h>
#include <conio.h>
#include <graph.h>
#include "imgdata.inc"
void
textattr(int newattr)
{
  _settextcolor(newattr & 15);         /* 0000xxxx */
  _setbkcolor((newattr >> 4) & 7);     /* 0xxx0000 */
}
int
main()
{
  char *ch;
  int attr;
  int pos;
  if (_setvideomode(_TEXTC80) == 0) {
    fputs("Error setting video mode", stderr);
    return 1;
  }
  /* draw the array */
  _settextposition(1, 1);              /* top left */
  /* print one line less than the 80x25 that's in there:
     80 x 24 x 2 = 3840 */

  for (pos = 0; pos < 3840; pos += 2) {
    ch = &IMAGEDATA[pos];              /* pointer assignment */
    attr = IMAGEDATA[pos + 1];
    textattr(attr);
```

```
    _outmem(ch, 1);
  }
  /* done */
  _settextposition(25, 1);             /* bottom left */
  textattr(0x0f);
  _outtext("Press any key to quit");
  getch();
  textattr(0x00);
  return 0;
}
```

Compile the program using the OpenWatcom C Compiler on FreeDOS, and you'll get a new program that displays this holiday message:

# Get started programming with DOS conio

By Jim Hall

One of the reasons so many DOS applications sported a text user interface (or TUI) is because it was so easy to do. The standard way to control **con**sole **i**nput and **o**utput (**conio**) was with the `conio` library for many C programmers. This is a de-facto standard library on DOS, which gained popularity as implemented by Borland's proprietary C compiler as `conio.h`. You can also find a similar `conio` implementation in TK Chia's IA-16 DOS port of the GNU C Compiler in the `libi86` library of non-standard routines. The library includes implementations of `conio.h` functions that mimic Borland Turbo C++ to set video modes, display colored text, move the cursor, and so on.

For years, FreeDOS included the OpenWatcom C Compiler in the standard distributions. OpenWatcom supports its own version of `conio`, implemented in `conio.h` for particular console input and output functions, and in `graph.h` to set colors and perform other manipulation. Because the OpenWatcom C Compiler has been used for a long time by many developers, this `conio` implementation is also quite popular. Let's get started with the OpenWatcom `conio` functions.

## Setting the video mode

Everything you do is immediately displayed on-screen via hardware. This is different from the `ncurses` library on Linux, where everything is displayed through terminal emulation. On DOS, everything is running on hardware. And that means DOS `conio` programs can easily access video modes and leverage screen regions in ways that are difficult using Linux `ncurses`.

To start, you need to set the *video mode*. On OpenWatcom, you do this with the `_setvideomode` function. This function takes one of several possible values, but for most programs that run in color mode in a standard 80x25 screen, use `_TEXTC80` as the mode.

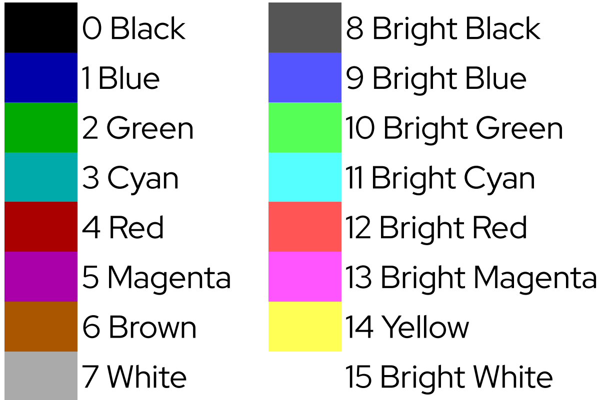
```
#include <conio.h>
#include <graph.h>
int
main()
{
  _setvideomode(_TEXTC80);
  …
```

When you're done with your program and ready to exit back to DOS, you should reset the video mode back to whatever values it had before. For that, you can use `_DEFAULTMODE` as the mode.

```
 _setvideomode(_DEFAULTMODE);
  return 0;
}
```

## Setting the colors

Every PC built after 1981's Color/Graphics Adapter supports [16 text colors and 8 background colors](). Background colors are addressed with color indices 0 through 7, and text colors can be any value from 0 to 15:




You can set both the text color and the color behind it. Use the `_settextcolor` function to set the text "foreground" color and `_setbkcolor` to set the text "background" color. For example, to set the colors to yellow text on a red background, you would use this pair of functions:

```
_settextcolor(14);
_setbkcolor(4);
```

## Positioning text

In conio, screen coordinates are always *row,col* and start with 1,1 in the upper-left corner. For a standard 80-column display with 25 lines, the bottom-right corner is 25,80.

Use the _settextposition function to move the cursor to a specific screen coordinate, then use _outtext to print the text you want to display. If you've set the colors, your text will use the colors you last defined, regardless of what's already on the screen.

For example, to print the text "FreeDOS" at line 12 and column 36 (which is more or less centered on the screen) use these two functions:

```
  _settextposition(12, 36);
  _outtext("FreeDOS");
```

Here's a small example program:

```
#include <conio.h>
#include <graph.h>
int
main()
{
    _setvideomode(_TEXTC80);
    _settextcolor(14);
    _setbkcolor(4);
    _settextposition(12, 36);
    _outtext("FreeDOS");
    getch();
    _setvideomode(_DEFAULTMODE);
    return 0;
}
```

Compile and run the program to see this output:

## Text windows

The trick to unleashing the power of `conio` is to leverage a feature of the PC video display where a program can control the video hardware by region. These are called text windows and are a really cool feature of `conio`.

A text window is just an area of the screen, defined as a rectangle starting at a particular *row*,*col* and ending at a different *row*,*col*. These regions can take up the whole screen or be as small as a single line. Once you define a window, you can clear it with a background color and position text in it.

To define a text window starting at row 5 and column 10, and extending to row 15 and column 70, you use the `_settextwindow` function like this:

```
_settextwindow(5, 10, 15, 70);
```

Now that you've defined the window, any text you draw in it uses 1,1 as the upper-left corner of the text window. Placing text at 1,1 will actually position that text at row 5 and column 10, where the window starts on the screen.

You can also clear the window with a background color. The `_clearscreen` function does double duty to clear either the full screen or just the window that's currently defined. To clear the entire screen, give the value `_GCLEARSCREEN` to the function. To clear just the window, use `_GWINDOW`. With either usage, you'll fill that region with whatever background color you last set. For example, to clear the whole screen with cyan (color 3) and a smaller text window with blue (color 1) you could use this code:

```
  _setbkcolor(3);
  _clearscreen(_GCLEARSCREEN);
  _settextwindow(5, 10, 15, 70);
  _setbkcolor(1);
  _clearscreen(_GWINDOW);
```

This makes it really easy to fill in certain areas of the screen. In fact, defining a window and filling it with color is such a common thing to do that I often create a function to do both at once. Many of my `conio` programs include some variation of these two functions to clear the screen or window:

```
#include <conio.h>
#include <graph.h>
void
clear_color(int fg, int bg)
{
  _settextcolor(fg);
  _setbkcolor(bg);
  _clearscreen(_GCLEARSCREEN);
}
void
textwindow_color(int top, int left, int bottom, int right, int fg, int bg)
{
  _settextwindow(top, left, bottom, right);
  _settextcolor(fg);
  _setbkcolor(bg);
  _clearscreen(_GWINDOW);
}
```

A text window can be any size, even a single line. This is handy to define a title bar at the top of the screen or a status line at the bottom of the screen. Again, I find this to be such a useful addition to my programs that I'll frequently write functions to do it for me:

```
#include <conio.h>
#include <graph.h>
#include <string.h>                    /* for strlen */
```

```
void
clear_color(int fg, int bg)
{
  …
}
void
textwindow_color(int top, int left, int bottom, int right, int fg, int bg)
{
  …
}
void
print_header(int fg, int bg, const char *text)
{
  textwindow_color(1, 1, 1, 80, fg, bg);
  _settextposition(1, 40 - (strlen(text) / 2));
  _outtext(text);
}
void
print_status(int fg, int bg, const char *text)
{
  textwindow_color(25, 1, 25, 80, fg, bg);
  _settextposition(1, 1);
  _outtext(text);
}
```

## Putting it all together

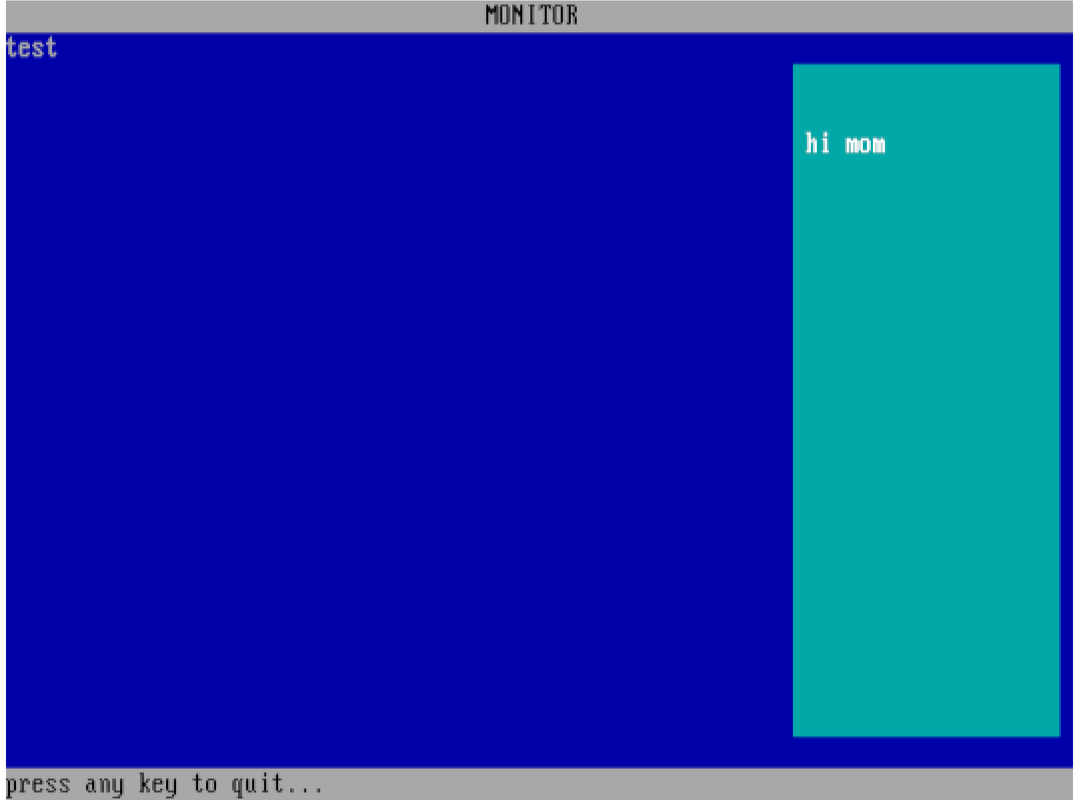
With this introduction to `conio`, and with the set of functions we've defined above, you can create the outlines of almost any program. Let's write a quick example that demonstrates how text windows work with `conio`. We'll clear the screen with a color, then print some sample text on the second line. That leaves room to put a title line at the top of the screen. We'll also print a status line at the bottom of the screen.

This is the basics of many kinds of applications. Placing a text window towards the right of the screen could be useful if you were writing a "monitor" program, such as part of a control system, like this:

```
#include <conio.h>
#include <graph.h>
int
main()
{
  _setvideomode(_TEXTC80);
  clear_color(7, 1);                     /* white on blue */
  _settextposition(2, 1);
```

```
  _outtext("test");
  print_header(0, 7, "MONITOR");       /* black on white */
  textwindow_color(3, 60, 23, 79, 15, 3);       /* br white on cyan */
  _settextposition(3, 2);
  _outtext("hi mom");
  print_status(0, 7, "press any key to quit...");       /* black on white */
  getch();
  _setvideomode(_DEFAULTMODE);
  return 0;
}
```

Having already written our own window functions to do most of the repetitive work, this program becomes very straightforward: clear the screen with a blue background, then print "test" on the second line. There's a header line and a status line, but the interesting part is in the middle where the program defines a text window near the right edge of the screen and prints some sample text. The `getch()` function waits for the user to press a key on the keyboard, useful when you need to wait until the user is ready:



We can change only a few values to completely change the look and function of this program. By setting the background to green and red text on a white window, we have the start of a solitaire card game:

```
#include <conio.h>
#include <graph.h>
int
main()
{
  _setvideomode(_TEXTC80);
  clear_color(7, 2);                   /* white on green */
  _settextposition(2, 1);
  _outtext("test");
  print_header(14, 4, "SOLITAIRE");    /* br yellow on red */
  textwindow_color(10, 10, 17, 22, 4, 7);       /* red on white */
  _settextposition(3, 2);
  _outtext("hi mom");
  print_status(7, 6, "press any key to quit...");       /* white on brown */
  getch();
  _setvideomode(_DEFAULTMODE);
  return 0;
}
```

You could add other code to this sample program to print card values and suits, place cards on top of other cards, and other functionality to create a complete game. But for this demo, we'll just draw a single "card" displaying some text:

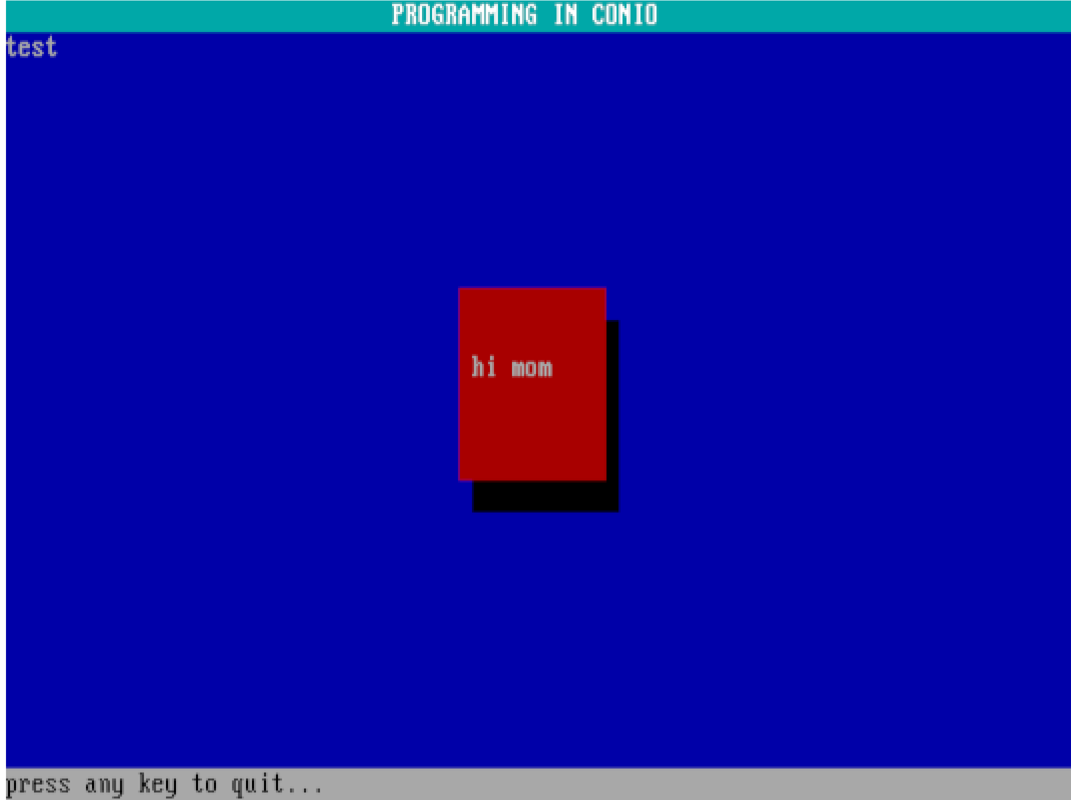
You can create other effects using text windows. For example, before drawing a message window, you could first draw a black window that's offset by one row and one column. The text window will appear to create a shadow over that area of the screen to the user. And we can do it all by changing only a few values in our sample program:

```
#include <conio.h>
#include <graph.h>
int
main()
{
  _setvideomode(_TEXTC80);
  clear_color(7, 1);                    /* white on blue */
  _settextposition(2, 1);
  _outtext("test");
  print_header(15, 3, "PROGRAMMING IN CONIO");  /* br white on cyan */
  textwindow_color(11, 36, 16, 46, 7, 0);       /* shadow */
  textwindow_color(10, 35, 15, 45, 7, 4);       /* white on red */
  _settextposition(3, 2);
  _outtext("hi mom");

  print_status(0, 7, "press any key to quit...");       /* black on white */
  getch();
```

```
  _setvideomode(_DEFAULTMODE);
  return 0;
}
```

You often see this "shadow" effect used in DOS programs as a way to add some visual flair:




The DOS `conio` functions can do much more than I've shown here, but with this introduction to `conio` programming, you can create various practical and exciting applications. Direct screen access means your programs can be more interactive than a simple command-line utility that scrolls text from the bottom of the screen. Leverage the flexibility of `conio` programming and make your next DOS program a great one.

# How to program in C on FreeDOS

By Jim Hall

When I first started using DOS, I enjoyed writing games and other interesting programs using BASIC, which DOS included. Much later, I learned the C programming language.

I immediately loved working in C! It was a straightforward programming language that gave me a ton of flexibility for writing useful programs. In fact, much of the FreeDOS core utilities are written in C and Assembly.

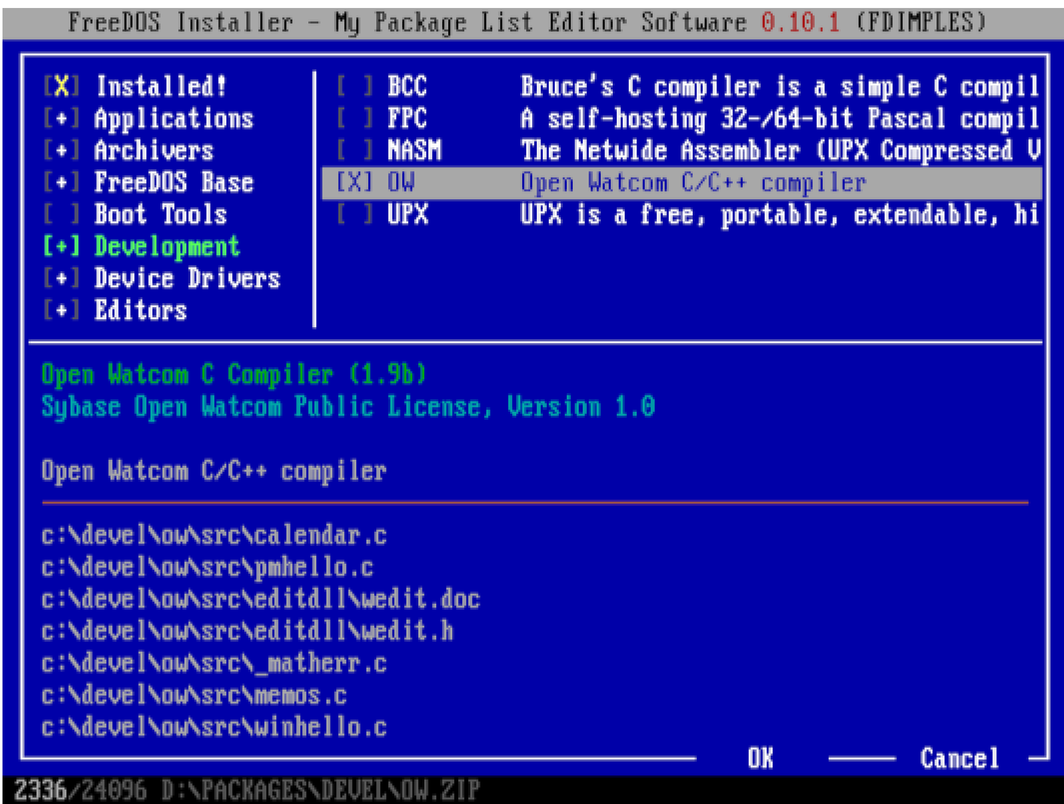
So it's probably not surprising that FreeDOS 1.3 RC4 includes a C compiler—along with other programming languages. The FreeDOS 1.3 RC4 LiveCD includes two C compilers—Bruce's C compiler (a simple C compiler) and the OpenWatcom C compiler. On the Bonus CD, you can also find DJGPP (a 32-bit C compiler based on GNU GCC) and the IA-16 port of GCC (requires a '386 or better CPU to compile, but the generated programs can run on low-end systems).

Programming in C on FreeDOS is basically the same as C programming on Linux, with two exceptions:

1. **You need to remain aware of how much memory you use.** Linux allows programs to use lots of memory, but FreeDOS is more limited. Thus, DOS programs used one of four memory models (large, medium, compact, and small) depending on how much memory they needed.
2. **You can directly access the console.** On Linux, you can create *text-mode* mode programs that draw to the terminal screen using a library like *ncurses*. But DOS allows programs to access the console and video hardware. This provides a great deal of flexibility in writing more interesting programs.

I like to write my C programs in the IA-16 port of GCC, or OpenWatcom, depending on what program I am working on. The OpenWatcom C compiler is easier to install since it's only a single package. That's why we provide OpenWatcom on the FreeDOS LiveCD, so you can

install it automatically if you choose to do a "Full installation including applications and games" when you install FreeDOS 1.3 RC4. If you opted to install a "Plain DOS system," then you'll need to install the OpenWatcom C compiler afterward, using the FDIMPLES package manager.




## DOS C programming

You can find documentation and library guides on the OpenWatcom project website to learn all about the unique DOS C programming libraries provided by the OpenWatcom C compiler. To briefly describe a few of the most useful functions:

From `conio.h`:

- `int getch(void)`—Get a single keystroke from the keyboard
- `int getche(void)`—Get a single keystroke from the keyboard, and echo it

From `graph.h`:

- `_settextcolor(short color)`—Sets the color when printing text
- `_setbkcolor(short color)`—Sets the background color when printing text

- `_settextposition(short y, short x)`—Move the cursor to row y and column x
- `_outtext(char _FAR *string)`—Print a string directly to the screen, starting at the current cursor location

DOS only supports sixteen text colors and eight background colors. You can use the values 0 (Black) to 15 (Bright White) to specify the text colors, and 0 (Black) to 7 (White) for the background colors:

- **0**—Black
- **1**—Blue
- **2**—Green
- **3**—Cyan
- **4**—Red
- **5**—Magenta
- **6**—Brown
- **7**—White
- **8**—Bright Black
- **9**—Bright Blue
- **10**—Bright Green
- **11**—Bright Cyan
- **12**—Bright Red
- **13**—Bright Magenta
- **14**—Yellow
- **15**—Bright White

## A fancy "Hello world" program

The first program many new developers learn to write is a program that just prints "Hello world" to the user. We can use the DOS "conio" and "graphics" libraries to make this a more interesting program and print "Hello world" in a rainbow of colors.

In this case, we'll iterate through each of the text colors, from 0 (Black) to 15 (Bright White). As we print each line, we'll indent the next line by one space. When we're done, we'll wait for the user to press any key, then we'll reset the screen and exit.

You can use any text editor to write your C source code. I like using a few different editors, including FreeDOS Edit and Freemacs, but more recently I've been using the FED

editor because it provides *syntax highlighting*, making it easier to see keywords, strings, and variables in my program source code.

```
#include <stdio.h>
#include <conio.h>
#include <graph.h>

int
main()
{
    short color;
    short row = 1, col = 1;

    _setvideomode(_TEXTC80);

    for (color = 0; color <= 15; color++) {
        _settextposition(row++, col++);
        _settextcolor(color);
        _setbkcolor(color ? 0 : 7);
        _outtext("Hello world");
    }

    getch();
    _setvideomode(_DEFAULTMODE);

    return 0;
}
- a-d c:\src\test.c - line 24 - col 2  - 0x-- (--)
```

Before you can compile using OpenWatcom, you'll need to set up the DOS environment variables so OpenWatcom can find its support files. The OpenWatcom C compiler package includes a setup batch file that does this for you, as `\DEVEL\OW\OWSETENV.BAT`. Run this batch file to automatically set up your environment for OpenWatcom.

Once your environment is ready, you can use the OpenWatcom compiler to compile this "Hello world" program. I've saved my C source file as `TEST.C`, so I can type `WCL TEST.C` to compile and link the program into a DOS executable, called `TEST.EXE`. In the output messages from OpenWatcom, you can see that `WCL` actually calls the OpenWatcom C Compiler (`WCC`) to compile, and the OpenWatcom Linker (`WLINK`) to perform the object linking stage:

```
C:\SRC>wcl test.c
Open Watcom C/C++16 Compile and Link Utility Version 1.9
Portions Copyright (c) 1988-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
        wcc TEST.C
DOS/4GW Protected Mode Run-time  Version 1.97
Copyright (c) Rational Systems, Inc. 1990-1994
Open Watcom C16 Optimizing Compiler Version 1.9
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
TEST.C: 24 lines, included 1375, 0 warnings, 0 errors
Code size: 96
        wlink @__wcl__.lnk
DOS/4GW Protected Mode Run-time  Version 1.97
Copyright (c) Rational Systems, Inc. 1990-1994
Open Watcom Linker Version 1.9
Portions Copyright (c) 1985-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
loading object files
searching libraries
creating a DOS executable
C:\SRC>_
```

OpenWatcom prints some extraneous output that may make it difficult to spot errors or warnings. To tell the compiler to suppress most of these extra messages, use the `/Q` ("Quiet") option when compiling:

```
C:\SRC>wcl /q test.c
DOS/4GW Protected Mode Run-time  Version 1.97
Copyright (c) Rational Systems, Inc. 1990-1994
DOS/4GW Protected Mode Run-time  Version 1.97
Copyright (c) Rational Systems, Inc. 1990-1994
C:\SRC>_
```

If you don't see any error messages when compiling the C source file, you can now run your DOS program. This "Hello world" example is `TEST.EXE`. Enter `TEST` on the DOS command line to run the new program, and you should see this very pretty output:

C is a very efficient programming language that works well for writing programs on limited-resource systems like DOS. There's lots more that you can do by programming in C on DOS. If you're new to the C language, you can learn C yourself by following along in our [Writing FreeDOS Programs in C]() self-paced ebook on the FreeDOS website, and the accompanying "how-to" video series on the [FreeDOS YouTube channel]().

# The basics of programming with C

By Seth Kenlon

In 1972, Dennis Ritchie was at Bell Labs, where a few years earlier, he and his fellow team members invented Unix. After creating an enduring OS (still in use today), he needed a good way to program those Unix computers so that they could perform new tasks. It seems strange now, but at the time, there were relatively few programming languages; Fortran, Lisp, Algol, and B were popular but insufficient for what the Bell Labs researchers wanted to do. Demonstrating a trait that would become known as a primary characteristic of programmers, Dennis Ritchie created his own solution. He called it C, and nearly 50 years later, it's still in widespread use.

## Why you should learn C

Today, there are many languages that provide programmers more features than C. The most obvious one is C++, a rather blatantly named language that built upon C to create a nice object-oriented language. There are many others, though, and there's a good reason they exist. Computers are good at consistent repetition, so anything predictable enough to be built into a language means less work for programmers. Why spend two lines recasting an `int` to a `long` in C when one line of C++ (`long x = long(n);`) can do the same?

And yet C is still useful today.

First of all, C is a fairly minimal and straightforward language. There aren't very advanced concepts beyond the basics of programming, largely because C is literally one of the foundations of modern programming languages. For instance, C features arrays, but it doesn't offer a dictionary (unless you write it yourself). When you learn C, you learn the building blocks of programming that can help you recognize the improved and elaborate designs of recent languages.

## Programming and development

Because C is a minimal language, your applications are likely to get a boost in performance that they wouldn't see with many other languages. It's easy to get caught up in the race to the bottom when you're thinking about how fast your code executes, so it's important to ask whether you *need* more speed for a specific task. And with C, you have less to obsess over in each line of code, compared to, say, Python or Java. C is fast. There's a good reason the Linux kernel is written in C.

Finally, C is easy to get started with, especially if you're running Linux. You can already run C code because Linux systems include the GNU C library (`glibc`). To write and build it, all you need to do is install a compiler, open a text editor, and start coding.

## Getting started with C

If you're running Linux, you can install a C compiler using your package manager. On Fedora or RHEL:

```
$ sudo dnf install gcc
```

On Debian and similar:

```
$ sudo apt install build-essential
```

On macOS, you can install Homebrew and use it to install GCC:

```
$ brew install gcc
```

On Windows, you can install a minimal set of GNU utilities, GCC included, with MinGW.

Verify you've installed GCC on Linux or macOS:

```
$ gcc --version
gcc (GCC) x.y.z
Copyright (C) 20XX Free Software Foundation, Inc.
```

On Windows, provide the full path to the EXE file:

```
PS> C:\MinGW\bin\gcc.exe --version
gcc.exe (MinGW.org GCC Build-2) x.y.z
Copyright (C) 20XX Free Software Foundation, Inc.
```

## C syntax

C isn't a scripting language. It's compiled, meaning that it gets processed by a C compiler to produce a binary executable file. This is different from a scripting language like Bash or a hybrid language like Python.

In C, you create *functions* to carry out your desired task. A function named `main` is executed by default.

Here's a simple "hello world" program written in C:

```
#include <stdio.h>

int main() {
  printf("Hello world");
  return 0;
}
```

The first line includes a *header file*, essentially free and very low-level C code that you can reuse in your own programs, called `stdio.h` (standard input and output). A function called `main` is created and populated with a rudimentary print statement. Save this text to a file called `hello.c`, then compile it with GCC:

```
$ gcc hello.c --output hello
```

Try running your C program:

```
$ ./hello
Hello world$
```

## Return values

It's part of the Unix philosophy that a function "returns" something to you after it executes: nothing upon success and something else (an error message, for example) upon failure. These return codes are often represented with numbers (integers, to be precise): 0 represents nothing, and any number higher than 0 represents some non-successful state.

There's a good reason Unix and Linux are designed to expect silence upon success. It's so that you can always plan for success by assuming no errors nor warnings will get in your way when executing a series of commands. Similarly, functions in C expect no errors by design.

You can see this for yourself with one small modification to make your program appear to fail:

```
include <stdio.h>

int main() {
  printf("Hello world");
  return 1;
}
```

Compile it:

```
$ gcc hello.c --output failer
```

Now run it using a built-in Linux test for success. The && operator executes the second half of a command only upon success. For example:

```
$ echo "success" && echo "it worked"
success
it worked
```

The `||` test executes the second half of a command upon *failure*.

```
$ ls blah || echo "it did not work"
ls: cannot access 'blah': No such file or directory
it did not work
```

Now try your program, which does *not* return 0 upon success; it returns 1 instead:

```
$ ./failer && echo "it worked"
String is: hello
```

The program executed successfully, yet did not trigger the second command.

## Variables and types

In some languages, you can create variables without specifying what *type* of data they contain. Those languages have been designed such that the interpreter runs some tests against a variable in an attempt to discover what kind of data it contains. For instance, Python knows that `var=1` defines an integer when you create an expression that adds `var` to something that is obviously an integer. It similarly knows that the word `world` is a string when you concatenate `hello` and `world`.

C doesn't do any of these investigations for you; you must define your variable type. There are several types of variables, including integers (int), characters (char), float, and Boolean.

You may also notice there's no string type. Unlike Python and Java and Lua and many others, C doesn't have a string type and instead sees strings as an array of characters.

Here's some simple code that establishes a `char` array variable, and then prints it to your screen using printf along with a short message:

```
#include <stdio.h>

int main() {
   char var[6] = "hello";
   printf("Your string is: %s\r\n",var);
```

You may notice that this code sample allows six characters for a five-letter word. This is because there's a hidden terminator at the end of the string, which takes up one byte in the array. You can run the code by compiling and executing it:

```
$ gcc hello.c --output hello
$ ./hello
hello
```

## Functions

As with other languages, C functions take optional parameters. You can pass parameters from one function to another by defining the type of data you want a function to accept:

```
#include <stdio.h>

int printmsg(char a[]) {
   printf("String is: %s\r\n",a);
}

int main() {
   char a[6] = "hello";
   printmsg(a);
   return 0;
}
```

The way this code sample breaks one function into two isn't very useful, but it demonstrates that `main` runs by default and how to pass data between functions.

## Conditionals

In real-world programming, you usually want your code to make decisions based on data. This is done with *conditional* statements, and the `if` statement is one of the most basic of them.

To make this example program more dynamic, you can include the `string.h` header file, which contains code to examine (as the name implies) strings. Try testing whether the string passed to the `printmsg` function is greater than 0 by using the `strlen` function from the `string.h` file:

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

int printmsg(char a[]) {
  size_t len = strlen(a);
  if ( len > 0) {
    printf("String is: %s\r\n",a);
  } }

int main() {
   char a[6] = "hello";
   printmsg(a);
   return 1;
}
```

As implemented in this example, the sample condition will never be untrue because the string provided is always "hello," the length of which is always greater than 0. The final touch to this humble re-implementation of the `echo` command is to accept input from the user.

## Command arguments

The `stdio.h` file contains code that provides two arguments each time a program is launched: a count of how many items are contained in the command (`argc`) and an array containing each item (`argv`). For example, suppose you issue this imaginary command:

```
$ foo -i bar
```

The `argc` is three, and the contents of `argv` are:

- `argv[0] = foo`
- `argv[1] = -i`
- `argv[2] = bar`

Creative Commons Attribution Share-alike 4.0

Can you modify the example C program to accept `argv[2]` as the string instead of defaulting to `hello`?

## Imperative programming

C is an imperative programming language. It isn't object-oriented, and it has no class structure. Using C can teach you a lot about how data is processed and how to better manage the data you generate as your code runs. Use C enough, and you'll eventually be able to write libraries that other languages, such as Python and Lua, can use.

To learn more about C, you need to use it. Look in `/usr/include/` for useful C header files, and see what small tasks you can do to make C useful to you.

# C Programming Cheat Sheet

By Jim Hall

C is a straightforward compiled programming language. Other programming languages borrow concepts from C, which makes C a great starting point if you want to learn programming languages such as Lua, C++, Java, or Go.

## Basics

Include header files first, then define your global variables, then write your program.

```
/* comment to describe the program */
#include <stdio.h>
/* definitions */
int main(int argc, char **argv) {
   /* variable declarations */
   /* program statements */
}
```

## Variables

Variable names can contain uppercase or lowercase letters (A to Z, or a to z), or numbers (0 to 9), or an underscore (_). Cannot start with a number.

| | |
|---|---|
| `int` | Integer values (-1, 0, 1, 2, …) |
| `char` | Character values, such as letters |
| `float` | Floating point numbers (0.0, 1.1, 4.5, or 3.141) |
| `double` | Double precision numbers, like float but bigger |

## Functions

Indicate the function type and name followed by variables inside parentheses. Put your function statements inside curly braces.

```
int celsius(int fahr) {
  int cel;
  cel = (fahr - 32) * 5 / 9;
  return cel;
}
```

Allocate memory with **malloc**. Resize with **realloc**. Use **free** to release.

```
int *array;
int *newarray;

arr = (int *) malloc(sizeof(int) * 10);
if (arr == NULL) {
   /* fail */
}

newarray = (int *) realloc(array,
sizeof(int) * 20);

if (newarray == NULL) {
   /* fail */
}
arr = newarray;

free(arr);
```

 CC BY-SA 4.0

# C Programming Cheat Sheet

By Jim Hall

## Binary operators

| | |
|---|---|
| `a & b` | Bitwise AND (1 if both bits are 1) |
| `a | b` | Bitwise OR (1 if either bits are 1) |
| `a ^ b` | Bitwise XOR (1 if bits differ) |
| `a<<n` | Shift bits to the left |
| `a>>n` | Shift bits to the right |

## Assignment shortcuts

| | | |
|---|---|---|
| `a += b;` | Addition | `a = a + b;` |
| `a -= b;` | Subtraction | `a = a - b;` |
| `a *= b;` | Multiplication | `a = a * b;` |
| `a /= b;` | Division | `a = a / b;` |
| `a %= b;` | Modulo | `a = a % b;` |

## Useful functions <stdio.h>

| | |
|---|---|
| `stdin` | Standard input (from user or another program) |
| `stdout` | Standard output (print) |
| `stderr` | Dedicated error output |

```
FILE *fopen(char *filename, char *mode);

size_t fread(void *ptr, size_t size,
size_t nitems, FILE *stream);
int fclose(FILE *stream);

int puts(char *string);
int printf(char *format, …);
int fprintf(FILE *stream, char *format);
int sprintf(char *string, char *format);

int getc(FILE *stream);
int putc(int ch, FILE *stream);

int getchar();
int putchar(int ch);
```

## Useful functions <stdlib.h>

```
void *malloc(size_t size);
void *realloc(void *ptr, size_t newsize);

void free(void *ptr);

void qsort(void *array, size_t nitems,
size_t size, int (*compar)(void *a, void
*b));

void *bsearch(void *key, void *array,
size_t nitems, size_t size, int (*compar)
(void *a, void *b));

void srand(unsigned int seed);

void rand();
```

Always test for **NULL** when allocating memory with ***malloc*** or ***realloc***.

If you ***malloc*** or ***realloc***, you should also ***free***. But only free memory once.

 **CC BY-SA 4.0**

# Appendix: How C has grown

By Jim Hall

The C programming language will turn fifty years old in 2022. Yet despite its long history, C remains one of the top "most-used" programming languages in many "popular programming languages" surveys. For example, check out the TIOBE Index, which tracks the popularity of different programming languages. Many Linux applications are written in C, such as the GNOME desktop.

I interviewed Brian Kernighan, co-author (with Dennis Ritchie) of *The C Programming Language* book, to learn more about the C programming language and its history.

**Where did the C programming language come from?**

C is an evolution of a sequence of languages intended for system programming—that is, writing programs like compilers, assemblers, editors, and ultimately operating systems. The Multics project at MIT, with Bell Labs as a partner, planned to write everything in a high-level language (a new idea at the time, roughly 1965). They were going to use IBM's PL/1, but it was very complicated, and the promised compilers didn't arrive in time.

After a brief flirtation with a subset called EPL (by Doug McIlroy of Bell Labs), Multics turned to BCPL, a much simpler and cleaner language designed and implemented by Martin Richards of Cambridge, who I think was visiting MIT at the time. When Ken Thompson started working on what became Unix, he created an even simpler language, based on BCPL, that he called B. He implemented it for the PDP-7 used for the first proto-Unix system in 1969.

BCPL and B were both "typeless" languages; that is, they had only one data type, integer. The DEC PDP-11, which arrived on the scene in about 1971 and was the computer for the first real Unix implementation, supported several data types, notably 8-bit bytes as well as 16-bit integers. For that, a language that also supported several data types was a better fit. That's the origin of C.

**How was C used within Bell Labs and the early versions of Unix?**

C was originally used only on Unix, though after a while, there were also C compilers for other machines and operating systems. Mostly it was used for system-programming applications, which covered quite a spectrum of interesting areas, along with a lot of systems for managing operations of AT&T's telephone network.

**What was the most interesting project written in C at Bell Labs?**

Arguably, the most interesting, memorable, and important C program was the Unix operating system itself. The first version of Unix in 1971 was in PDP-11 assembly language, but by the time of the fourth edition, around 1973, it was rewritten in C. That was truly crucial since it meant that the operating system (and all its supporting software) could be ported to a different kind of computer basically by recompiling everything. Not quite that simple in practice, but not far off.

**You co-authored *The C Programming Language* book with Dennis Ritchie. How did that book come about, and how did you and Dennis collaborate on the book?**

I had written a tutorial on Ken Thompson's B language to help people get started with it. I upgraded that to a tutorial on C when it became available. And after a while, I twisted Dennis's arm to write a C book with me. Basically, I wrote most of the tutorial material, except for the system call chapter, and Dennis had already written the reference manual, which was excellent. Then we worked back and forth to smooth out the tutorial parts; the reference manual stayed pretty much the same since it was so well done from the beginning. The book was formatted with the *troff* formatter, one of many tools on Unix, and I did most of the formatting work.

**When did C become a thing that other programmers outside of Bell Labs used for their work?**

I don't really remember well at this point, but I think C mostly followed along with Unix for the first half dozen years or so. With the development of compilers for other operating systems, it began to spread to other systems besides Unix. I don't recall when we realized that C and Unix were having a real effect, but it must have been in the mid to late 1970s.

**Why did C become such an influential programming language?**

The primary reason in the early days was its association with Unix, which spread rapidly. If you used Unix, you wrote in C. Later on, C spread to computers that might not necessarily run Unix, though many did because of the portable C compiler that Steve Johnson wrote. The

workstation market, with companies like Sun Microsystems, MIPS (which became SGI), and others, was enabled by the combination of Unix and C. The IBM PC came somewhat later, about 1982, and C became one of the standard languages, under MS-DOS and then Windows. And today, most Internet of Things (IoT) devices will use C.

**C remains a popular programming language today, some 50 years after its creation. Why has C remained so popular?**

I think C hit a sweet spot with efficiency and expressiveness. In earlier times, efficiency really mattered since computers were slow and had limited memory compared to what we are used to today. C was very efficient, in the sense that it could be compiled into efficient machine code, and it was simple enough that it was easy to see how to compile it. At the same time, it was very expressive, easy to write, and compact. No other language has hit that kind of spot quite so well, at least in my humble but correct opinion.

**How has the C programming language grown or changed over the years?**

C has grown modestly, I guess, but I haven't paid much attention to the evolving C standards. There are enough changes that code written in the 1980s needs a bit of work before it will compile, but it's mostly related to being honest about types. Newer features like complex numbers are perhaps useful, but not to me, so I can't make an informed comment.

**What programming problems can be solved most easily in C?**

Well, it's a good language for anything, but today, with lots of memory and processing power, most programmers are well served by languages like Python that take care of memory management and other more high-level constructs. C remains a good choice for lower levels where squeezing cycles and bytes still matter.

**C has influenced other programming languages, including C++, Java, Go, and Rust. What are your thoughts on these other programming languages?**

Almost every language is in some ways a reaction to its predecessors. To over-simplify a fair amount, C++ adds mechanisms to control access to information, so it's better than C for really large programs. Java is a reaction to the perceived complexity of C++. Go is a reaction to the complexity of C++ and the restrictions of Java. Rust is an attempt to deal with memory management issues in C (and presumably C++) while coming close to C's efficiency.

They all have real positive attributes, but somehow no one is ever quite satisfied, so there will always be more languages that, in their turn, react to what has gone before. At the same time, the older languages, for the most part, will remain around because they do their job well, and

there's an embedded base where they are perfectly fine, and it would be infeasible to reimplement in something newer.

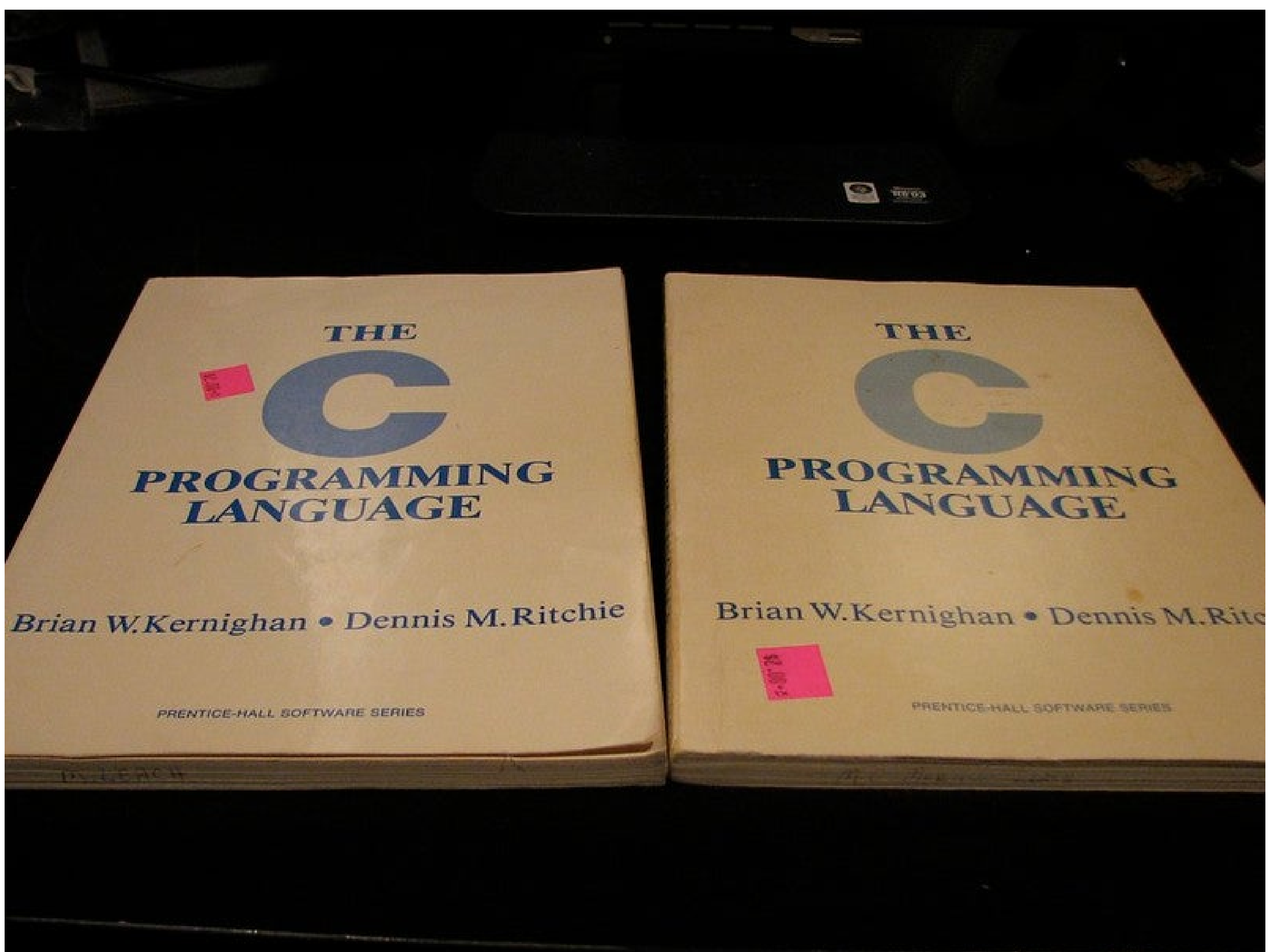
---

Thanks to Brian for sharing this great history of the C programming language!

# The monumental impact of C

By Matthew Broberg

C is the original general-purpose programming language. The Season 3 finale of the Command Line Heroes podcast explores C's origin story in a way that showcases the longevity and power of its design. It's a perfect synthesis of all the languages discussed throughout the podcast's third season and this series of articles.




Creative Commons Attribution Share-alike 4.0 

C is such a fundamental language that many of us forget how much it has changed. Technically a "high-level language," in the sense that it requires a compiler to be runnable, it's as close to assembly language as people like to get these days (outside of specialized, low-memory environments). It's also considered to be the language that made nearly all languages that came after it possible.

## The path to C began with failure

While the myth persists that all great inventions come from highly competitive garage dwellers, C's story is more fit for the Renaissance period.

In the 1960s, Bell Labs in suburban New Jersey was one of the most innovative places of its time. Jon Gertner, author of *The idea factory*, describes the culture of the time marked by optimism and the excitement to solve tough problems. Instead of monetization pressures with tight timelines, Bell Labs offered seemingly endless funding for wild ideas. It had a research and development ethos that aligns well with today's open leadership principles. The results were significant and prove that brilliance can come without the promise of VC funding or an IPO.

The challenge back then was terminal sharing: finding a way for lots of people to access the (very limited number of) available computers. Before there was a scalable answer for that, and long before we had a shell like Bash, there was the Multics project. It was a hypothetical operating system where hundreds or even thousands of developers could share time on the same system. This was a dream of John McCarty, creator of Lisp and the term artificial intelligence (AI), as I recently explored.

Joy Lisi Ranken, author of *A people's history of computing in the United States*, describes what happened next. There was a lot of public interest in driving forward with Multics' vision of more universally available timesharing. Academics, scientists, educators, and some in the broader public were looking forward to this computer-powered future. Many advocated for computing as a public utility, akin to electricity, and the push toward timesharing was a global movement.

Up to that point, high-end mainframes topped out at 40-50 terminals per system. The change of scale was ambitious and eventually failed, as Warren Toomey writes in IEEE Spectrum:

> "Over five years, AT&T invested millions in the Multics project, purchasing a GE-645 mainframe computer and dedicating to the effort many of the top researchers at the company's renowned Bell Telephone Laboratories—including Thompson and Ritchie, Joseph F. Ossanna, Stuart Feldman, M. Douglas McIlroy, and the late Robert Morris. But the new system was too ambitious, and it fell troublingly behind schedule. In the end, AT&T's corporate leaders decided to pull the plug."

Bell Labs pulled out of the Multics program in 1969. Multics wasn't going to happen.

## The fellowship of the C

Funding wrapped up, and the powerful GE645 mainframe was assigned to other tasks inside Bell Labs. But that didn't discourage everyone.

Among the last holdouts from the Multics project were four men who felt passionately tied to the project: Ken Thompson, Dennis Ritchie, Doug McIlroy, and J.F. Ossanna. These four diehards continued to muse and scribble ideas on paper. Thompson and Ritchie developed a game called Space Travel for the PDP-7 minicomputer. While they were working on that, Thompson started implementing all those crazy hand-written ideas about filesystems they'd developed among the wreckage of Multics.

That's worth emphasizing: Some of the original filesystem specifications were written by hand and then programmed on what was effectively a toy compared to the systems they were using to build Multics. Wikipedia's Ken Thompson page dives deeper into what came next:

> "While writing Multics, Thompson created the Bon programming language. He also created a video game called Space Travel. Later, Bell Labs withdrew from the MULTICS project. In order to go on playing the game, Thompson found an old PDP-7 machine and rewrote Space Travel on it. Eventually, the tools developed by Thompson became the Unix operating system: Working on a PDP-7, a team of Bell Labs researchers led by Thompson and Ritchie, and including Rudd Canaday, developed a hierarchical file system, the concepts of computer processes and device files, a command-line interpreter, pipes for easy inter-process communication, and some small utility programs. In 1970, Brian Kernighan suggested the name 'Unix,' in a pun on the name 'Multics.' After initial work on Unix, Thompson decided that Unix needed a system programming language and created B, a precursor to Ritchie's C."

As Walter Toomey documented in the IEEE Spectrum article mentioned above, Unix showed promise in a way the Multics project never materialized. After winning over the team and doing a lot more programming, the pathway to Unix was paved.

## Getting from B to C in Unix

Thompson quickly created a Unix language he called B. B inherited much from its predecessor BCPL, but it wasn't enough of a breakaway from older languages. B didn't know data types, for starters. It's considered a typeless language, which meant its "Hello World" program looked like this:

```
main( ) {
extrn a, b, c;
putchar(a); putchar(b); putchar(c); putchar('!*n');
}
a 'hell';
b 'o, w';
c 'orld';
```

Even if you're not a programmer, it's clear that carving up strings four characters at a time would be limiting. It's also worth noting that this text is considered the original "Hello World" from Brian Kernighan's 1972 book, *A tutorial introduction to the language B* (although that claim is not definitive).

Typelessness aside, B's assembly-language counterparts were still yielding programs faster than was possible using the B compiler's threaded-code technique. So, from 1971 to 1973, Ritchie modified B. He added a "character type" and built a new compiler so that it didn't have to use threaded code anymore. After two years of work, B had become C.

## The right abstraction at the right time

C's use of types and ease of compiling down to efficient assembly code made it the perfect language for the rise of minicomputers, which speak in bytecode. B was eventually overtaken by C. Once C became the language of Unix, it became the de facto standard across the budding computer industry. Unix was *the* sharing platform of the pre-internet days. The more people wrote C, the better it got, and the more it was adopted. It eventually became an open standard itself. According to the Brief history of C programming language:

> "For many years, the de facto standard for C was the version supplied with the Unix operating system. In the summer of 1983 a committee was established to create an ANSI (American National Standards Institute) standard that would define the C language. The standardization process took six years (much longer than anyone reasonably expected)."

How influential is C today? A quick review reveals:

- Parts of all major operating systems are written in C, including macOS, Windows, Linux, and Android.
- The world's most prolific databases, including DB2, MySQL, MS SQL, and PostgreSQL, are written in C.
- Many programming-language specifics begun in C, including Python, Go, Perl's core interpreter, and the R statistical language.

Decades after they started as scrappy outsiders, Thompson and Ritchie are praised as titans of the programming world. They shared 1983's Turing Award, and in 1998, received the National Medal of Science for their work on the C language and Unix.

But Doug McIlroy and J.F. Ossanna deserve their share of praise, too. All four of them are true Command Line Heroes.

# How to write a good C main function

By Erik O'Shaughnessy

I know, Python and JavaScript are what the kids are writing all their crazy "apps" with these days. But don't be so quick to dismiss C—it's a capable and concise language that has a lot to offer. If you need speed, writing in C could be your answer. If you are looking for job security and the opportunity to learn how to hunt down null pointer dereferences, C could also be your answer! In this article, I'll explain how to structure a C file and write a C main function that handles command line arguments like a champ.

**Me**: a crusty Unix system programmer.

**You**: someone with an editor, a C compiler, and some time to kill.

*Let's do this.*

## A boring but correct C program

A C program starts with a **main()** function, usually kept in a file named **main.c**.

```
/* main.c */
int main(int argc, char *argv[]) {
}
```

This program *compiles* but doesn't *do* anything.

```
$ gcc main.c
$ ./a.out -o foo -vv
$
```

Correct and boring.

## Main functions are unique

The **main()** function is the first function in your program that is executed when it begins executing, but it's not the first function executed. The *first* function is **_start()**, which is typically provided by the C runtime library, linked in automatically when your program is compiled. The details are highly dependent on the operating system and compiler toolchain, so I'm going to pretend I didn't mention it.

The **main()** function has two arguments that traditionally are called **argc** and **argv** and return a signed integer. Most Unix environments expect programs to return **0** (zero) on success and **-1** (negative one) on failure.

| Argument | Name | Description |
|---|---|---|
| argc | Argument count | Length of the argument vector |
| argv | Argument vector | Array of character pointers |

The argument vector, **argv**, is a tokenized representation of the command line that invoked your program. In the example above, **argv** would be a list of the following strings:

```
argv = [ "/path/to/a.out", "-o", "foo", "-vv" ];
```

The argument vector is guaranteed to always have at least one string in the first index, **argv[0]**, which is the full path to the program executed.

## Anatomy of a main.c file

When I write a **main.c** from scratch, it's usually structured like this:

```
/* main.c */
/* 0 copyright/licensing */
/* 1 includes */
/* 2 defines */
/* 3 external declarations */
/* 4 typedefs */
/* 5 global variable declarations */
/* 6 function prototypes */
int main(int argc, char *argv[]) {
/* 7 command-line parsing */
}
/* 8 function declarations */
```

I'll talk about each of these numbered sections, except for zero, below. If you have to put copyright or licensing text in your source, put it there.

Another thing I won't talk about adding to your program is comments.

```
"Comments lie."
- A cynical but smart and good looking programmer.
```

Instead of comments, use meaningful function and variable names.

Appealing to the inherent laziness of programmers, once you add comments, you've doubled your maintenance load. If you change or refactor the code, you need to update or expand the comments. Over time, the code mutates away from anything resembling what the comments describe.

If you have to write comments, do not write about *what* the code is doing. Instead, write about *why* the code is doing what it's doing. Write comments that you would want to read five years from now when you've forgotten everything about this code. And the fate of the world is depending on you. *No pressure*.

## 1. Includes

The first things I add to a **main.c** file are includes to make a multitude of standard C library functions and variables available to my program. The standard C library does lots of things; explore header files in **/usr/include** to find out what it can do for you.

The **#include** string is a C preprocessor (cpp) directive that causes the inclusion of the referenced file, in its entirety, in the current file. Header files in C are usually named with a **.h** extension and should not contain any executable code; only macros, defines, typedefs, and external variable and function prototypes. The string **<header.h>** tells cpp to look for a file called **header.h** in the system-defined header path, usually **/usr/include**.

```
/* main.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libgen.h>
#include <errno.h>
#include <string.h>
#include <getopt.h>
#include <sys/types.h>
```

This is the minimum set of global includes that I'll include by default for the following stuff:

| #include File | Stuff It Provides |
|---|---|
| stdio | Supplies FILE, stdin, stdout, stderr, and the fprint() family of functions |
| stdlib | Supplies malloc(), calloc(), and realloc() |
| unistd | Supplies EXIT_FAILURE, EXIT_SUCCESS |
| libgen | Supplies the basename() function |
| errno | Defines the external errno variable and all the values it can take on |
| string | Supplies memcpy(), memset(), and the strlen() family of functions |
| getopt | Supplies external optarg, opterr, optind, and getopt() function |
| sys/types | Typedef shortcuts like uint32_t and uint64_t |

## 2. Defines

```
/* main.c */
<...>
#define OPTSTR "vi:o:f:h"
#define USAGE_FMT  "%s [-v] [-f hexflag] [-i inputfile] [-o outputfile] [-h]"
#define ERR_FOPEN_INPUT  "fopen(input, r)"
#define ERR_FOPEN_OUTPUT "fopen(output, w)"
#define ERR_DO_THE_NEEDFUL "do_the_needful blew up"
#define DEFAULT_PROGNAME "george"
```

This doesn't make a lot of sense right now, but the **OPTSTR** define is where I will state what command line switches the program will recommend. Consult the **getopt(3)** man page to learn how **OPTSTR** will affect **getopt()**'s behavior.

The **USAGE_FMT** define is a **printf()**-style format string that is referenced in the **usage()** function.

I also like to gather string constants as **#defines** in this part of the file. Collecting them makes it easier to fix spelling, reuse messages, and internationalize messages, if required.

Finally, use all capital letters when naming a **#define** to distinguish it from variable and function names. You can run the words together if you want or separate words with an underscore; just make sure they're all upper case.

## 3. External declarations

```
/* main.c */
<...>
```

```
extern int errno;
extern char *optarg;
extern int opterr, optind;
```

An **extern** declaration brings that name into the namespace of the current compilation unit (aka "file") and allows the program to access that variable. Here we've brought in the definitions for three integer variables and a character pointer. The **opt** prefaced variables are used by the **getopt()** function, and **errno** is used as an out-of-band communication channel by the standard C library to communicate why a function might have failed.

## 4. Typedefs

```
/* main.c */
<...>
typedef struct {
  int           verbose;
  uint32_t      flags;
  FILE         *input;
  FILE         *output;
} options_t;
```

After external declarations, I like to declare **typedefs** for structures, unions, and enumerations. Naming a **typedef** is a religion all to itself; I strongly prefer a **_t** suffix to indicate that the name is a type. In this example, I've declared **options_t** as a **struct** with four members. C is a whitespace-neutral programming language, so I use whitespace to line up field names in the same column. I just like the way it looks. For the pointer declarations, I prepend the asterisk to the name to make it clear that it's a pointer.

## 5. Global variable declarations

```
/* main.c */
<...>
int dumb_global_variable = -11;
```

Global variables are a bad idea and you should never use them. But if you have to use a global variable, declare them here and be sure to give them a default value. Seriously, *don't use global variables*.

## 6. Function prototypes

```
/* main.c */
<...>
void usage(char *progname, int opt);
int  do_the_needful(options_t *options);
```

As you write functions, adding them after the **main()** function and not before, include the function prototypes here. Early C compilers used a single-pass strategy, which meant that every symbol (variable or function name) you used in your program had to be declared before you used it. Modern compilers are nearly all multi-pass compilers that build a complete symbol table before generating code, so using function prototypes is not strictly required. However, you sometimes don't get to choose what compiler is used on your code, so write the function prototypes and drive on.

As a matter of course, I always include a **usage()** function that **main()** calls when it doesn't understand something you passed in from the command line.

## 7. Command line parsing

```
/* main.c */
<...>
int main(int argc, char *argv[]) {
    int opt;
    options_t options = { 0, 0x0, stdin, stdout };
    opterr = 0;
    while ((opt = getopt(argc, argv, OPTSTR)) != EOF)
       switch(opt) {
           case 'i':
              if (!(options.input = fopen(optarg, "r")) ){
                 perror(ERR_FOPEN_INPUT);
                 exit(EXIT_FAILURE);
                 /* NOTREACHED */
              }
              break;
           case 'o':
              if (!(options.output = fopen(optarg, "w")) ){
                 perror(ERR_FOPEN_OUTPUT);
                 exit(EXIT_FAILURE);
                 /* NOTREACHED */
              }
              break;

           case 'f':
              options.flags = (uint32_t )strtoul(optarg, NULL, 16);
```

```
            break;
         case 'v':
            options.verbose += 1;
            break;
         case 'h':
         default:
            usage(basename(argv[0]), opt);
            /* NOTREACHED */
            break;
      }
   if (do_the_needful(&options) != EXIT_SUCCESS) {
      perror(ERR_DO_THE_NEEDFUL);
      exit(EXIT_FAILURE);
      /* NOTREACHED */
   }
   return EXIT_SUCCESS;
}
```

OK, that's a lot. The purpose of the **main()** function is to collect the arguments that the user provides, perform minimal input validation, and then pass the collected arguments to functions that will use them. This example declares an **options** variable initialized with default values and parse the command line, updating **options** as necessary.

The guts of this **main()** function is a **while** loop that uses **getopt()** to step through **argv** looking for command line options and their arguments (if any). The **OPTSTR #define** earlier in the file is the template that drives **getopt()**'s behavior. The **opt** variable takes on the character value of any command line options found by **getopt()**, and the program's response to the detection of the command line option happens in the **switch** statement.

Those of you paying attention will now be questioning why **opt** is declared as a 32-bit **int** but is expected to take on an 8-bit **char**? It turns out that **getopt()** returns an **int** that takes on a negative value when it gets to the end of **argv**, which I check against **EOF** (the *End of File* marker). A **char** is a signed quantity, but I like matching variables to their function return values.

When a known command line option is detected, option-specific behavior happens. Some options have an argument, specified in **OPTSTR** with a trailing colon. When an option has an argument, the next string in **argv** is available to the program via the externally defined variable **optarg**. I use **optarg** to open files for reading and writing or converting a command line argument from a string to an integer value.

There are a couple of points for style here:

- Initialize **opterr** to 0, which disables **getopt** from emiting a **?**.
- Use **exit(EXIT_FAILURE);** or **exit(EXIT_SUCCESS);** in the middle of **main()**.
- **/* NOTREACHED */** is a lint directive that I like.
- Use **return EXIT_SUCCESS;** at the end of functions that return **int**.
- Explicitly cast implicit type conversions.

The command line signature for this program, were it compiled, looks something like this:

```
$ ./a.out -h
a.out [-v] [-f hexflag] [-i inputfile] [-o outputfile] [-h]
```

In fact, that's what **usage()** emits to **stderr** once compiled.

## 8. Function declarations

```
/* main.c */
<...>
void usage(char *progname, int opt) {
   fprintf(stderr, USAGE_FMT, progname?progname:DEFAULT_PROGNAME);
   exit(EXIT_FAILURE);
   /* NOTREACHED */
}
int do_the_needful(options_t *options) {
   if (!options) {
     errno = EINVAL;
     return EXIT_FAILURE;
   }
   if (!options->input || !options->output) {
     errno = ENOENT;
     return EXIT_FAILURE;
   }
  /* XXX do needful stuff */
   return EXIT_SUCCESS;
}
```

Finally, I write functions that aren't boilerplate. In this example, function **do_the_needful()** accepts a pointer to an **options_t** structure. I validate that the **options** pointer is not **NULL** and then go on to validate the **input** and **output** structure members. **EXIT_FAILURE** returns if either test fails and, by setting the external global variable **errno** to a conventional error code, I signal to the caller a general reason. The convenience function **perror()** can be used by the caller to emit human-readable-ish error messages based on the value of **errno**.

Functions should almost always validate their input in some way. If full validation is expensive, try to do it once and treat the validated data as immutable. The **usage()** function validates the **progname** argument using a conditional assignment in the **fprintf()** call. The **usage()** function is going to exit anyway, so I don't bother setting **errno** or making a big stink about using a correct program name.

The big class of errors I am trying to avoid here is de-referencing a **NULL** pointer. This will cause the operating system to send a special signal to my process called **SYSSEGV**, which results in unavoidable death. The last thing users want to see is a crash due to **SYSSEGV**. It's much better to catch a **NULL** pointer in order to emit better error messages and shut down the program gracefully.

Some people complain about having multiple **return** statements in a function body. They make arguments about "continuity of control flow" and other stuff. Honestly, if something goes wrong in the middle of a function, it's a good time to return an error condition. Writing a ton of nested **if** statements to just have one return is never a "good idea."™

Finally, if you write a function that takes four or more arguments, consider bundling them in a structure and passing a pointer to the structure. This makes the function signatures simpler, making them easier to remember and not screw up when they're called later. It also makes calling the function slightly faster, since fewer things need to be copied into the function's stack frame. In practice, this will only become a consideration if the function is called millions or billions of times. Don't worry about it if that doesn't make sense.

## Wait, you said no comments!?!!

In the **do_the_needful()** function, I wrote a specific type of comment that is designed to be a placeholder rather than documenting the code:

```
/* XXX do needful stuff */
```

When you are in the zone, sometimes you don't want to stop and write some particularly gnarly bit of code. You'll come back and do it later, just not now. That's where I'll leave myself a little breadcrumb. I insert a comment with a **XXX** prefix and a short remark describing what needs to be done. Later on, when I have more time, I'll grep through source looking for **XXX**. It doesn't matter what you use, just make sure it's not likely to show up in your codebase in another context, as a function name or variable, for instance.

## Putting it all together

OK, this program *still* does almost nothing when you compile and run it. But now you have a solid skeleton to build your own command line parsing C programs.

```
/* main.c - the complete listing */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libgen.h>
#include <errno.h>
#include <string.h>
#include <getopt.h>
#define OPTSTR "vi:o:f:h"
#define USAGE_FMT  "%s [-v] [-f hexflag] [-i inputfile] [-o outputfile] [-h]"
#define ERR_FOPEN_INPUT  "fopen(input, r)"
#define ERR_FOPEN_OUTPUT "fopen(output, w)"
#define ERR_DO_THE_NEEDFUL "do_the_needful blew up"
#define DEFAULT_PROGNAME "george"
extern int errno;
extern char *optarg;
extern int opterr, optind;
typedef struct {
  int           verbose;
  uint32_t      flags;
  FILE         *input;
  FILE         *output;
} options_t;
int dumb_global_variable = -11;
void usage(char *progname, int opt);
int  do_the_needful(options_t *options);
int main(int argc, char *argv[]) {
    int opt;
    options_t options = { 0, 0x0, stdin, stdout };
    opterr = 0;
    while ((opt = getopt(argc, argv, OPTSTR)) != EOF)
       switch(opt) {
           case 'i':
              if (!(options.input = fopen(optarg, "r")) ){
                 perror(ERR_FOPEN_INPUT);
                 exit(EXIT_FAILURE);
                 /* NOTREACHED */
              }
              break;
           case 'o':
              if (!(options.output = fopen(optarg, "w")) ){
                 perror(ERR_FOPEN_OUTPUT);
                 exit(EXIT_FAILURE);
```

```
                /* NOTREACHED */
             }
             break;

          case 'f':
             options.flags = (uint32_t )strtoul(optarg, NULL, 16);
             break;
          case 'v':
             options.verbose += 1;
             break;
          case 'h':
          default:
             usage(basename(argv[0]), opt);
             /* NOTREACHED */
             break;
       }
    if (do_the_needful(&options) != EXIT_SUCCESS) {
       perror(ERR_DO_THE_NEEDFUL);
       exit(EXIT_FAILURE);
       /* NOTREACHED */
    }
    return EXIT_SUCCESS;
}
void usage(char *progname, int opt) {
   fprintf(stderr, USAGE_FMT, progname?progname:DEFAULT_PROGNAME);
   exit(EXIT_FAILURE);
   /* NOTREACHED */
}
int do_the_needful(options_t *options) {
   if (!options) {
     errno = EINVAL;
     return EXIT_FAILURE;
   }
   if (!options->input || !options->output) {
     errno = ENOENT;
     return EXIT_FAILURE;
   }
   /* XXX do needful stuff */
   return EXIT_SUCCESS;
}
```

Now you're ready to write C that will be easier to maintain.