# Linux User's Group of Davis
# C Programming Basics, Part I

Speaker: Mark K. Kim

November 16, 1999

## 1 What is C?

We are going to talk about a programming language known as *C*. This is a very popular programming language, especially on the UNIX-based platforms, including Linux. As matter of a fact, most UNIX-based systems are created entirely on C, including Linux. *C*

Since you should know what you're learning, you should know that there are several "dialects" of C. You should also know which dialects there are and which one you are learning, so I list here the three main dialects of C:

1. "Classic" C (a.k.a. "Old" C, K&R C)

   This is the original C dialect as was created by Dennis M. Ritchie and Brian W. Kernighan. Although this dialect has no official name, it is often referred to as *"Classic" C*. Other popular names for this dialect includes *"Old" C* and *K&R C*.

   This dialect is old, yet it is still used widely today to write programs that need to be compiled under multiple platforms, including the platforms with older C compilers (newer compilers can compile both Classic C and ANSI C programs.) An example of a such program is VIM.

2. ANSI/ISO C (a.k.a. ANSI C, C89, C90, "New" C)

   This is the standard (and most popular) dialect of C, and was created to fix problems with, and add features to, Classic C.

   This is the dialect we will talk about.

3. C9X

   This is the new C standard being worked on by the Working Group 14 (JTC1/SC22/WG14). They are trying to improve upon ANSI/ISO C by adding new features and also implementing some C++ features.

   This dialect has not yet been standardized so we will not talk much about this upcoming standard. For more information on C9X, please visit the WG14 homepage, at `http://www.dkuug.dk/JTC1/SC22/WG14/`.

For our talk, we will concentrate on ANSI/ISO C since that is the current standard.

## 2 Let's C...

### 2.1 A very simple program—coding it, compiling it, and running it

Let's look at a very simple program to get us started. I call this the "Hello, LUGOD!" program. It prints out a line of text,

```
Hello, LUGOD!
```

to the screen. To create this program, we open up a text editor (vi, emacs, pico, notepad, edit...) and type the following code:

```
#include <stdio.h>

int main() {
   printf("Hello, LUGOD!\n");

   return 0;
}
```

We will see what the program does a bit later. For now, we want to see if it works. Save the program as `hello.c` and exit.

Now we need to *compile* the program in order to create a file we can run.[1] Compiling is creating a runnable program, known as the binary or the executable, from a text file called the *sourcecode*. *compile* *sourcecode*

We compile our sourcecode (`hello.c`) using a C compiler. In our case, we use a program named `gcc`:

```
gcc hello.c -o hello
```

*`gcc`* is the compiler we're using, *`hello.c`* is the file you just created, and *`-o hello`* tells `gcc` that we want our program to be named *`hello`*.[2] If the compilation was successful, you can run the program. Type *`hello`* and it will print out

```
Hello, LUGOD!
```

But if `gcc` gives error messages, open up `hello.c` and check for errors. Then fix the problem and compile again.

Obviously, if the error message is

```
gcc: Command not found.
```

then you do not have `gcc` installed on your computer. Please consult with your nearest Linux user (or DJGPP user, if you are using DOS) for help... but please do this after the meeting to not disturb others!

## 2.2 How this "very simple program" works

### 2.2.1 `printf`

This C program has a lot of lines that cannot be understood right now. So let's take one line at a time, starting with the most obvious:

```
printf("Hello, LUGOD!\n");
```

This is the line of code that prints "`Hello, LUGOD!`" to the screen. `printf` is the name of the command that tells the computer to print to the screen, and `"Hello, LUGOD!\n"` is what gets printed to the screen. *printf*

You may notice that "`\n`" is not printed. `\n` does not get printed physically, but it functions to move the cursor to the next line. Kind of like hitting enter at the keyboard. `\n` is often called the *newline* character, because printing that character moves the cursor to the next line. *newline*

Try the program after removing "`\n`." You'll see that the output is missing a line compared to before.

You need the semicolon (;) at the end of the `printf` statement because every *statement* in C *statement*

[1] You may wonder why you need to create a compiled program, especially if you're a fanatic script programmer. Scripted programs have the advantage that you do not need to compile it (which takes some time,) but compiled programs has the advantage that it runs fast once compiled; both programming languages are useful, but for different reasons, depending on how you want your program to behave.

[2] by default, `gcc` creates a program named `a.out` (or, in case of DOS, `a.exe`.)

ends with a semicolon.[3] (Commands are considered to be statements.) Requiring semicolons to come after each statement allows the programmer to separate a line of code into multiple lines as s/he chooses—a statement need not be written all on one line because the end-of-line does not end a statement, but the semicolon does. The opposite is also true—a programmer can choose to put multiple statements on one line, because the end-of-lines do not end the statements, but the semicolons.

By the way, *commands* are called *functions* in C, and that is how we will refer to the "commands" from here on. There are a couple reasons why "commands" are called functions: *functions*

- For one, every command has a function, whether it's a computer command or a command to a dog. In our case, the `printf` has the function of printing to the screen.
- And two, the syntax of a command in C looks a lot like a math function—just compare a C function like "`printf("...")`" with a math function like "$\sin(\theta)$."

Now that you know why commands are called functions, you won't forget it!

We will come back to `printf` later and see how you can also print numbers, which is different from printing a text. We will also see how you can use `printf` to print text in a desired format.

### 2.2.2 `int main() {... return 0;}`

Now that you know what `printf` does, let's see what "`int main() {... return 0;}`" does. (Notice I am now able to print the almost-entire program in one line because the semicolons terminate statements, not the end-of-lines.)

The purpose of `main` is this: The program must start somewhere, and it must end somewhere; for C, everything starts where `main` begins and ends where `main` ends. The syntax of `main` looks like this: *main*

```
int main() {
   ...

   return N;
}
```

where "`...`" is your program and *N* is some number, usually zero.

You can see that you need the braces to mark where `main` (your program) begins and ends. But why do you need the `int`, the parenthesis, and the `return` statement, and what is *N* used for?

We cannot talk about this now, but we will talk about this later. Like I mentioned before, not everything can be understood right now. We will come back to this. I promise.

### 2.2.3 `#include`

Now we look at the `#include` statement.

This also requires some more knowledge of C before we can understand why we need it, but we can have a short explanation right here.

The explanation is this: The reason we require #include <stdio.h> is because there is a file named `stdio.h`, and this file contains information about the `printf` function, and we need this information in order to use `printf` in our program. If we don't include it, we get an error message during the compilation process.[4]

These files you `#include` are called *header files* because these files usually go on the "head" (the beginning) of your program. *#include* *header files*

In the future, whenever you need to use a function, and if you are not sure which header file you need to `#include`, you can refer to the `man` page, section 3 (the subroutines section.) For example, if I want to know which header file I need to `#include` in order to use `printf`, I type

[3]Semicolons are *not* statement separators as is in Pascal, but a statement terminator. Semicolons are important and are easily forgotten by those new to C. We will see where else semicolons are required later.

[4]Actually, we're *supposed* to get error messages. Some compilers include them for you by default, including the version of `gcc` on my Linux.

*man 3 printf*

and I get the following information:

```
PRINTF(3)           Linux Programmer's Manual           PRINTF(3)

NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf, vfprintf,
       vsprintf, vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       ...
```

See where it says "`#include <stdio.h>`?" That's the header file you need to include in your C program.

We will get back to header files later, to learn about *other* header files, and to see what functions we can use by including *them*.

# 3 Let's write a game!

Now that we know how the "Hello, LUGOD!" program works, let's write a computer program. We can learn about new features of C as we write this program.

Since many programmers start out as wanna-be-game-programmers, let's write a simple text-based adventure game.

In case some of you are not familiar with what text files are, may I take a moment to explain what it is? A text-based game tells you your status—where you are, what the surrounding looks like, what you see, etc.—and it asks you what you want to do. You then tell it where you want to go or what you want to do, and you get a response back from the game. If you have ever played MUD (Multi-User Dungeon) or BBS (Bulletin-Board System) games before, then you already have an idea of what it is like, except you generally don't play text-based games with other people as you normally do in MUD or BBS.[5]

## 3.1 Startup credit notice

When you start a game, the first thing you usually notice is the startup credits, kind of like this one from XPilot:

```
Copyright © 1991-96 by Bjørn Stabell, Ken Ronny Schouten & Bert Gijsbers.
XPilot 3.6.2 comes with ABSOLUTELY NO WARRANTY; for details see the
provided LICENSE file.
```

We will have something similar in our game.

We first need to decide what we want to print—like the program name, copyright notice, etc. How about tentatively naming our program "LUGOD Adventures?" And we'll print a copyright notice right below it. And maybe our webpage address below *that*. Kind of like this:

```
LUGOD Adventures
Copyright (c)1999 Linux Users Group of Davis
http://www.lugod.org/
```

You already know how to do this. Let's see what the program looks like:

[5] In that aspect, I suppose you can call text-based games *SUD*—Single-User Dungeon games.

```
#include <stdio.h>

int main() {
   printf("LUGOD Adventures\n");
   printf("Copyright (c)1999 Linux Users Group of Davis\n";
   printf("http://www.lugod.org/\n");

   return 0;
}
```

Good. Let's move onto the next step: The starting point of our game world.

### 3.2 The starting point of the game—a pizza place

A pizza place, I suppose, is a good place to start our text game.

Let's describe this place from the second person's point of view, as is common in text-based games:

> It is a corner room at a pizza place. It is filled with many people, all sitting at separate tables, and all attentively listening to someone in front who appears to be a speaker. There are two people on your table, one wearing a penguin suit and one wearing a red hat. How tacky, you think to yourself. There is also a medium-sized pizza in front of you.

While we're at it, it'd be nice to "suddenly wake up" at this place as you always do in text games. So let's write this out in our program (program shortened to save space):

```
...
   printf("You suddenly wake up in a corner room at a pizza place.\n");
   printf("It is filled with many people, all sitting at separate tables,\n");
   printf("and all attentively listening to someone in front\n");
   printf("who appears to be a speaker.  There are two people on your table,\n");
   printf("one wearing a penguin suit and one wearing a red hat.  How tacky,\n");
   printf("you think to yourself.  There is also a medium-sized pizza in\n");
   printf("front of you.\n");

   return 0;
}
```

Now the game needs to ask the player what s/he wants to do. We can do this in several ways. Here are a couple of options we have:

1. Display an options menu, let the player pick one of the options.

2. Create different commands the player can make use of, and let the player type them in. For example, allow the player to type in a command like *`pickup pizza`*.

Option #1 is probably easier to code, so let's go with that.

First, display some of our options to the screen, and ask the player what s/he wants to do:

```
...
   printf("1. Ask the penguin what this place is called.\n");
   printf("2. Ask the red hat what this meeting is about.\n");
   printf("3. Pickup the pizza and throw it at the penguin.\n");
   printf("4. Pickup the pizza and throw it at the red hat.\n");
   printf("5. Pickup the pizza and throw it at the speaker.\n");
```

```
    printf("\n");   /* Extra line for spacing */

    printf("What do you want to do? ");

    return 0;
}
```

I want you to notice three things in this piece of code:

1. There is a *comment* next to the 6th `printf` statement. You can create comments to make notes to yourself. Making notes to yourself is important because programmers often forget, after a long time, what a piece of code does. *comment*

   In C, "`/*`" begins a comment and "`*/`" ends a comment. You can put anything between these two character sequences, usually some useful information about some nearby code.

2. The 6th `printf` statement is used to print an extra line. This is for spacing reasons, because the output looks pretty crappy without it.

3. The last `printf` statement contains an extra space at the end of the text and contains no newline character (`"What do you want to do?␣"` instead of `"What do you want to do?\n"`). This is also for appearance reasons; we want our program to show

   ```
   What do you want to do? _
   ```

   instead of:

   ```
   What do you want to do?
   _
   ```

(where _ is the cursor.)

Now we need to get an input from the keyboard. A common way of doing this is using the `scanf` function. This is a two-step process: *scanf*

1. Create a *variable* you can store keystrokes in. A variable is a memory space you can store data in. You can create different types of variables (numbers, characters, etc.) depending on how you want to treat the data. *variable*

2. Read the keystroke(s), store them in the variable.

And here is how we do it:

```
...
int main() {
    int input;                          /* We create a variable here */
    ...
    printf("What do you want to do? ");   /* Ask what the player wants to do */
    scanf("%i", &input);                /* Read an integer, store in 'input' */

    return 0;
}
```

Let's analyze this for a little bit so we don't get lost or overwhelmed by what's coming next.

First, we have `int input`. This creates an *integer variable* named `input`.[6] *integer variable*

Then the `scanf` line reads an integer, and stores it in `input`. `"%i"` is the part that tells `scanf` to scan for an integer from whatever the programmer typed, then the *argument* following it (`input`) *argument*

[6]If you have programmed in a language that does not require you to create variables before using them, then you may wonder why you need to do this. Well... if you ever misspelled a variable name (thereby accidentally creating an unwanted variable) then you know how important it is to keep your variables under control. In C, you explicitly create variables so you don't end up with a misbehaving program because of a simple misspelling.

is where scanf stores the data. (You can use "%f" to read real numbers, "%c" to read a character, etc. We will also come back to this later.)

Now, why do you need & in front of input? You need & in front of input because you need & in front of a variable name if you want to allow the function (scanf in this case) to *write* to the variable. Without &, functions are not allowed to write to the variable. This is a way of controlling the behavior of a function without actually knowing how the function works—you can be sure the function will not change the value of a variable unless you allow it to.

There is an exception to this rule, however. If the variable is a *pointer* or an *array* (things we will talk about later,) then the functions can write to them without needing & in front of the variable name. There is a reason for this exception, but we cannot go into it right now. We'll get back to this. I promise.

*pointer*
*array*

## 3.3 Moving on to the next stage

Now we need to decide what to do once we know what the player has typed. We do it like so:

```
...
   scanf("%i", &input);      /* This line is from before */

   if(input == 1)                                   /* Player chooses 1 */
      printf("Penguin: This is Lamppost Pizza!\n");

   else if(input == 2)                              /* Player chooses 2 */
      printf("Red hat: This is a LUGOD meeting.\n");

   else if(input == 3 || input == 4)                /* Player chooses 3 or 4 */
      printf("Someone yells, ‘FOOD FIGHT!’\n");

   else if(input == 5)                              /* Player chooses 5 */
      printf("Your shirt catches on fire.\n");
      printf("You die.\n");
      printf("End of the game.\n");
   }

   else {
      printf("Invalid option!\n");
   }

   return 0;
}
```

We have a new statement of interest—the if statement. This statement (called a *keyword* because it is one of the few *key* components of the C language that requires a special syntax.)

*if*
*keyword*

The syntax of the if statement is as follows:

```
if(test-condition) statement;
```

or,

```
if(test-condition) statement;
else statement;
```

We can also put several of them together in a row (a common technique,) like so:

```
if(test-condition) statement;
else if(test-condition) statement;
else if(test-condition) statement;
...
```

since the `if` statement is also considered to be a statement.

The last usage is what we use in our program. We first check to see if our `input` is 1, then print a message accordingly. Then we check to see if our `input` is 2, then print a different message according to that. And so on.

Notice the following things:

1. A group of statements, grouped together by *braces* ({ and }) can be also used in place of a *statement*. So instead of *braces*

   ```
   if(test-condition) statement;
   ```

   you can have

   ```
   if(test-condition) {
      statement1;
      statement2;
      ...
   }
   ```

   This is okay—C treats a group of statements held together by braces as a single statement.

   This is what we use for the last `if` statement.

   Notice that you do not need a semicolon at the end of the braces. But if you want to put one, feel free—C won't complain.

2. There is something called the *test condition*. A test condition is something that allows one to see whether something is true or not, such as testing to see if our `input` variable is equivalent to some number. *test condition*

   We use the test condition "`input == some number`" to see if `input` is equivalent to some number. This is a very commonly used test condition. Note the usage of *double* equal signs (`==`) instead of *single* equal sign (`=`). The double equal sign is used for test conditions, and the single equal sign is used to assign a value to a variable; we will see this again later.

   Some other test conditions include the following:

   ```
   variable-or-number < variable-or-number    /* less than */
   variable-or-number <= variable-or-number   /* less than or equal to */
   variable-or-number > variable-or-number    /* greater than */
   variable-or-number >= variable-or-number   /* greater than or equal to */
   variable-or-number != variable-or-number   /* not equal to*/
   ```

   All of them should be familiar for most people except perhaps `!=`. This is different from `<>` used in many languages to represent a "not equal to" symbol.

3. We also use a compound test condition (the third `if` statement), where two test conditions (`input == 3` and `input == 4`) are held together by `||`.

   The `||` symbol is equivalent to the English word "or," and it is also equivalent to the math symbol $\cup$ (union).

   In terms of English, `if(input == 3 || input == 4)` is equivalent to the phrase "`if input` is equal to 3 *or* `input` is equal to 5." In terms of math, it is equivalent to "if $(input = 3) \cup (input = 4)$."

   If you are familiar with math symbols, you also know there is the $\cap$ (intersect). This is represented by the symbol `&&` and is equivalent to the English word "and."

   Don't forget—there are *two* vertical bars in a row (`||`) and *two* ampersands in a row (`&&`). Single vertical bar and a single ampersand mean something else—we'll get back to that later.

So our `if` statements print "`Penguin: This is Lamppost Pizza!`" if the player types *1* followed by enter, "`Red hat: This is a LUGOD meeting`" if the player types *2* followed by enter, "`Someone yells, 'FOOD FIGHT!'`" if the player types *3* or *4* followed by enter, and if the player types *5* followed by enter then it prints "`Your shirt catches on fire. You die. End of game.`"

Notice you *have* to type enter after choosing each option. It is possible to make the game continue without requiring the player to press enter, but doing this is different from one system to another thus there is no C standard for it. (Or rather, there is no C standard for it thus it is different from system to system.)

Now we need to decide where to take the story from here on. Let's move on.

## 3.4 Until the food fight...

As you can see from our game plot so far, the game doesn't go anywhere until the food fight starts—asking those two guys questions doesn't bring in much new plot, and throwing the pizza at the speaker kills the player. So we need to do the following three things:

1. If the player chooses options 1 or 2, keep displaying the same menu over and over until the player chooses one of the other three options.
2. If the player chooses options 3 or 4, lead the story somewhere else.
3. If the player chooses option 5, end the game.

Since #1 and #3 are probably the easiest to code, let's do those two first. These two new statements will come in handy:

- The `while` keyword *while*

  `while` forces a statement (or a group of statements in braces) to be executed until you want it to stop. The syntax is as follows:

  ```
  while(test-condition) statement;
  ```

  or,

  ```
  while(test-condition) {
     statement1;
     statement2;
     ...
  }
  ```

  The statement or the statements will run as long as *`test-condition`* is true.

- The `exit` function (`#include <stdlib.h>`) *exit*

  The `exit` function allows the program to exit in the middle of a program. The usage is as follows:

  ```
  exit(N);
  ```

  where *N* is some number, usually 1. We will see the significance of this number later, but not now. For now, we will just use 1.

And our resulting program is as follows:

```
...
   printf("front of you.\n");

   input = 0;              /* Assign 0 to input */
   while(input < 3) {
      printf("1. Ask the penguin what this place is called.\n");
```

```
      printf("2. Ask the red hat what this meeting is about.\n");
      printf("3. Pickup the pizza and throw it at the penguin.\n");
      printf("4. Pickup the pizza and throw it at the red hat.\n");
      printf("5. Pickup the pizza and throw it at the speaker.\n");

      printf("\n");       /* Extra line for spacing */

      printf("What do you want to do? ");       /* Ask the player what to do */
      scanf("%i", &input);                      /* Input a number */

      if(input == 1)                            /* Player types 1 */
         printf("Penguin: This is Lamppost Pizza!\n");
      else if(input == 2)                       /* Player types 2 */
         printf("Red hat: This is a LUGOD meeting.\n");
      else if(input == 3 || input == 4)         /* Player types 3 or 4*/
         printf("Someone yells 'FOOD FIGHT!'\n");
      else {                                    /* Player types 5 */
         printf("Your shirt catches on fire.\n");
         printf("You die.\n");
         printf("End of game.\n");
         printf("Serves you right.\n");
         exit(1);
      }
   } /* end while */

   /* Food-fight goes here */

   return 0;
}
```

Notice three things:

1. The menu and the keyboard input are enclosed inside a `while` statement. The menu is printed and the keyboard input is requested from the player until the player makes a choice of option 3 or above.

2. We *assign* 0 to `input` ("`input = 0`",) which causes `input` to contain the value zero, which causes the `while` loop to execute when `while` is run for the first time Without it, `input` may contain any value when the program runs, and the `while` loop may or may not get run. *assign*

   This may seem odd to some of you; in many languages, when you first create an integer variable, it is equal to zero. Not so in C—for a programming language, setting an integer variable to zero requires usage of extra computer time that may or may not be useful. C firmly believes that the programmer knows enough to set a variable to zero if s/he requires it. Indeed, we did not require it before, but now we do—by not having to initialize `input` to zero, our program ran a little bit faster as a C program than it would have as another language (about a microsecond or so, I suppose.)

3. There is an `exit` statement at the end of the last `if` statement. This may seem necessary now since the program ends whether you put this statement here or not; however, this allows us to place the food-fight code outside the `while` statement. If we not have the `exit` statement, we cannot place the food-fight code outside the `while` statement, at least not without using some `if` statement.

Great! Now compile and run the program. Not quite *that* interesting of a game, but it's improving.

## 3.5 The Food Fight!

Since our program is getting *really* big, at least big enough to fill an entire page to list the whole program, let's place the food-fight code in a separate function. Then we can save some space listing just the functions instead of the entire program.

Here is the syntax for creating a function:

```
return-type function-name(variable1-type variable1-name, variable2-type variable2-name, ...) {
   code...
}
```

For example, here is a simple function that returns the square of a number:

```
int get_square_of(int number) {
   int squared_number;

   squared_number = number * number;

   return squared_number;
}
```

To use it, you type something like

```
int n = get_square_of(5);
```

The result is the variable `n` getting the value 25.

Now, let's create a function that creates a food fight situation in our game world:

```
void foodfight() {
   int hp = 50;              /* Player's hit points */
   int scour = 0;            /* Number of hits */

   while(hp > 0) {           /* While I'm alive */
      int input;

      /* Display everyone's status */
      printf("Your hit points: %i\n", hp);
      printf("Your scour: %i hits\n", scour);
      printf("\n");          /* Extra line for spacing */

      /* Display the player's options */
      printf("1. Throw a pizza slice.\n");
      printf("2. Eat a pizza slice.\n");
      printf("3. Chicken out.\n");
      printf("\n");          /* Extra line for spacing */

      /* Ask the player what s/he wants to do */
      printf("What would you like to do? ");
      scanf("%i", &input);

      /* Player makes a choice */
      if(input == 1) {                 /* Throw a pizza */
         printf("You throw a pizza!\n");
         scour = scour + rand() / (RAND_MAX/2);
      }
      else if(input == 2) {
```

```
            printf("You eat a slice of pizza!\n");
            printf("HP up!\n");
            hp = hp + rand() / (RAND_MAX/10);
        }
        else if(input == 3) {
            printf("Someone throws a pizza pan from behind!\n");
            printf("A direct hit!\n");
            break;
        }
        else {
            printf("Invalid option: %i\n", input);
        }

        /* Someone attacks! */
        printf("Someone attacks the player!\n");
        hp = hp - rand() / (RAND_MAX / 8);
        if(hp < 0) break;
    } /* while(hp > 0) */

    printf("You faint into the darkness...\n");
    printf("Your scour: %i hits\n", scour);
    printf("The End.\n");
}
```

You should be able to understand everything here except the following:

1. This function's return type is `void`. This means the function does not return any value. *void*

   In some programming languages, functions that do not return any value are called *subroutines*. C does not make any distinction between subroutines and functions—subroutines are simply functions that do not return any value. Note that there is no `return` statement at the end of the function. *subroutines*

2. If you use `printf("%i",` *`integer-variable`*`)` then the program prints an integer to the screen. The usage is similar to `scanf`.

   You can also mix-and-match regular text and integers, like so:

   ```
   printf("I am %i years old.\n", 21);
   ```

   then `%i` is replaced by 21, printing "`I am 21 years old.`"

   So, basically, wherever `%i` appears, the value gets replaced by an integer argument following it.

   If you want to print several numbers, you can do that, too:

   ```
   printf("I am %i feet %i inches tall.\n", 5, 8);
   ```

   That prints out "`I am 5 feet 8 inches tall.`"

3. `rand` is a function that returns a random number between `0` and `RAND_MAX`. The numbers are carefully adjusted in our program to produce random numbers between the ranges we want. *rand*

4. `break` is a keyword used to break out of the `while` loop. The program continues where the `while` statement ends. *break*

Now you need to add a call to this function at the end of `main`:

```
...
    } /* end while */

    foodfight();

    return 0;
}
```

Also, you must place the function above `main` so C knows where the `foodfight` function is by the time you call the function in `main`.

Type it up, compile, and play!

## 3.6 Where to go from here

The game is as complete as one can make it at this point without making it too complicated. So we will stop working with this game at this point. From this point on, new capabilities of C will be introduced to you straight without any large scale examples like this one. We will use this game as a reference point of future examples, though.

By the way, you will need to practice programming in C if you want to become proficient in it. This program is as good a place to start as any—feel free to modify it and improve upon it. Maybe even add graphical interface to it? `:)`

On our next meeting, we will look at new C keywords and C features.