

C Bootcamp

CI Computer Girls

April 30, 2016

Hello World

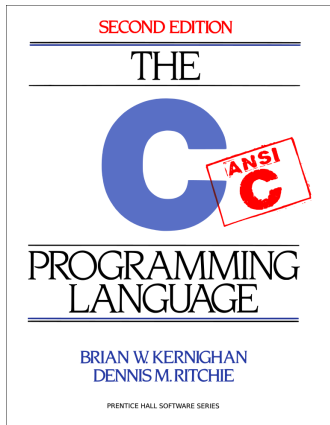


Figure 1: The bible.

- A C program consists of *functions* and *variables*.
- A function contains *statements* that specify the computing operations to be done.
- Variables store values to be used during computation.
- Normally you can name functions whatever you like, but every program must contain a function named `main`.

Hello World

```
1 #include <stdio.h>
2
3 main() {
4     printf("Hello, world!\n");
5 }
```

- In this example `printf` is a function that takes a *character string* as its argument.
- Copy the code above into an empty file `hello.c` in your `task1` directory (We'll help you find it.), and then from your terminal:

```
# cd ~/Desktop/bootcamp/task1
# gcc hello.c
# ./a.out
```

Prompts

From your terminal,

```
# cd ../task2
```

then open the file `prompt.c` in your text editor. You should see the following:

```
1  #include <stdio.h>
2
3  main() {
4      char name[40];
5      printf("Enter your name:\n");
6
7      // YOUR TASK: Prompt the user for their name say hello.
8
9  }
```

Prompts

```
1  #include <stdio.h>
2
3  main() {
4      char name[40];
5      printf("Enter your name:\n");
6
7      // YOUR TASK: Prompt the user for their name say hello.
8
9  }
```

- For this task, we'll make use of a new function

```
scanf(char* format, ...)
```

- `scanf` reads characters from your terminal, interprets them according to the `format` you provide (consult your cheatsheet), and stores the results in the remaining arguments.
- For example, to store a user-given string in `name`,

```
scanf("%s", name);
```

Prompts

```
1  #include <stdio.h>
2
3  main() {
4      char name[40];
5      printf("Enter your name:\n");
6
7      // YOUR TASK: Prompt the user for their name say hello.
8
9  }
```

- For example, to store a user-given string in `name`,
`scanf("%s", name);`
- Similarly, `printf` can be given format specifiers in its first argument and will print the rest of its arguments accordingly.
`printf("Goodbye %s", name);`
- Complete your task (Ask for your help if you're stuck!), and run your program.

Arguments

From your terminal,

```
# cd ../task3
```

then open the file `arguments.c` in your text editor. You should see the following:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[]) {
5      if (argc < 3) {
6          printf("Usage: %s <name> <integer>\n", argv[0]);
7          return -1;
8      }
9
10     // YOUR TASK: Read the user's name and an integer
11     // from command line arguments, then say hello
12     // to the user as many times as given by the integer.
13
14 }
```

Arguments

```
1 int main(int argc, char* argv[]) {  
2     ...  
3 }
```

- Note that our `main` has grown a little.
- The first `int` tells us that this function will return an integer.
- `int argc` and `char* argv[]` are parameters to `main`.
 - `char* argv[]` is an array of strings containing all the arguments we'll pass when we run our program. (More on that later.)
 - `int argc` is an integer indicating the length of `argv` or the number of strings contained within.

Arguments

```
1  int main(int argc, char* argv[]) {  
2      ...  
3  }
```

For example, if we invoke our program as follows:

```
# ./a.out CiComputerGirls 5
```

- Then `argc` contains the integer 3.
- `argv[0]` contains the string `"a.out"`.
- `argv[1]` contains the string `"CiComputerGirls"`.
- `argv[2]` contains the string `"5"`.

Arguments

```
1  int main(int argc, char* argv[]) {  
2      if (argc < 3) {  
3          printf("Usage: %s <name> <integer>\n", argv[0]);  
4          exit(-1);  
5      }  
6      ...  
7  }
```

- In the given code, we examine `argc` in the condition of our if-statement to ensure our program was passed the correct number of arguments.
- And if not, we print a helpful message and exit with an error code.
- Note that our helpful message prints the value of `argv[0]`. The first string in `argv` will always be the name of your program.

Arguments

```
1  int main(int argc, char* argv[]) {  
2      ...  
3  
4      // YOUR TASK: Read the user's name and an integer  
5      // from command line arguments, then say hello  
6      // to the user as many times as given by the integer.  
7  
8  }
```

To complete your task,

- Use the function `atoi` to convert the value of `argv[2]` into an `int`.

```
int atoi(char* s)
```

- `atoi` converts the string `s` into an `int`. For example,

```
int five = atoi("5");
```

- Then use a for- or while-loop to print your message as many times as needed.

Functions

From your terminal,

```
# cd ../task4
```

and open the file `functions.c` in your text editor.

```
1  #include <math.h>
2  #include <stdio.h>
3
4  // YOUR TASK: Implement the function get_population.
5
6  int get_population(int generation);
7
8  main() {
9      ...
10 }
```

Functions

Each function definition has the form

```
return-type function-name(argument declarations) {  
    declarations and statements  
}
```

Compare the above with our definition of the function `power` in `functions.c`.

```
1  int power(int base, int exp) {  
2      int result = 1;  
3  
4      int i;  
5      for (i = 0; i < exp; i++)  
6          result *= base;  
7  
8      return result;  
9  }
```

Functions

```
int get_population(int generation);
```

- Your implementation of `get_population` should conform to the specifications in the above declaration.
 - I.e., it should accept a single `int` as an argument and return an `int` to its caller.
- The rate at which your population grows is left up to you, but note that to work with our program shell, `get_population` should only increase with respect to its argument.
 - For a simple implementation consider that if each tribble can breed 2 more tribbles, then after 3 generations you have $2^3 = 8$ tribbles.
 - You can call `power` from within `get_population` to model this relationship.

Pointers

Now `cd` into the `task5` directory and open `pointers.c` in your text editor.

```
1  #include <stdio.h>
2
3  main() {
4      char *string1 = "This is the first string.";
5      char *string2 = "This is the second string.";
6
7      // YOUR TASK: Write a function `swap` that swaps the pointers
8      // in its first and second arguments. Then invoke your function
9      // with string1 and string2.
10
11     printf("string1 = %s and string2 = %s\n", string1, string2);
12 }
```

Pointers

```
1 | main() {  
2 |     char *string1 = "This is the first string.";  
3 |     ...  
4 | }
```

- A *pointer* is a variable that contains the address in memory of another variable.
- C has two operators for dealing with pointers:
 - `*` is the *dereferencing* operator. When applied to a pointer, it returns the value to which the pointer points.
 - `&` does the opposite of `*`. When applied to a variable, it returns the address at which the variable is stored.

Pointers

```
1 main() {  
2     char *string1 = "This is the first string.";  
3     ...  
4 }
```

- Note that the above code declares `string1` to be a pointer. It states that `string1` points to the location in memory of the beginning of the assigned string.
- Now note that `*string1` is a `char`. The `*` applies the dereferencing operator to `string1`, returning the first value at the location to which pointer points. In this case, `'T'`.

Pointers

To write your swap function, you must be aware of one more caveat:

- As in most languages, C passes arguments to functions by value, so there is no way for a called function to alter a variable in the calling function.
- Consider this function `swap` for swapping two integers:

```
1 // WRONG!
2 void swap(int x, int y) {
3     int temp;
4
5     temp = x;
6     x = y;
7     y = temp;
8 }
```

- Just like Java, this function will not work as intended.
- `x` and `y` are swapped within the scope of our function `swap`, but since they were passed only by value to the function, they will not be swapped in any code that calls our `swap`.

Pointers

To obtain the desired effect, instead of writing a function that takes the values of the variables to be swapped, we write a function that takes the pointers to their values in memory:

```
1 void swap(int *px, int *py) {  
2     int temp;  
3  
4     temp = *px;    // temp gets the value to which px points.  
5     *px = *py;     // The value at px gets the value at py.  
6     *py = temp;    // The value to which py points gets temp.  
7 }
```

We can invoke this `swap` like so:

```
1 int a = 5;  
2 int b = 10;  
3 swap(&a, &b);  
4 // Now a == 10, and b == 5.
```

Pointers

```
1 void swap(int *px, int *py) {  
2     int temp;  
3  
4     temp = *px;    // temp gets the value to which px points.  
5     *px = *py;     // The value at px gets the value at py.  
6     *py = temp;    // The value to which py points gets temp.  
7 }  
8  
9 int a = 5;  
10 int b = 10;  
11 swap(&a, &b);  
12 // Now a == 10, and b == 5.
```

- Note that to swap two `int` our function accepts two `int*`.
- We use `&` to access the address at which our values are stored. In other words, to access *pointers* to our variables `a` and `b`.

Pointers

```
1 main() {  
2     char* string1 = "This is the first string.";  
3     char* string2 = "This is the second string.";  
4  
5     // YOUR TASK: Write a function `swap` that swaps the pointers  
6     // in its first and second arguments. Then invoke your function  
7     // with string1 and string2.  
8  
9     printf("string1 = %s and string2 = %s\n", string1, string2);  
10 }
```

- Since a string is already a pointer to a `char` (or a `char*`), your `swap` should take two pointers to `char` pointers (or two `char**`).
- Implement `swap` and test your program. (And ask for help if you're stuck!)

Structs

Now `cd` into the `task6` directory and open `structs.c` in your text editor.

```
1  typedef struct {
2      char* name;
3      int health;
4      int* weapon_statuses; // An array whose elements indicate the
5                           // status of each weapon.
6                           // A value >= 100 indicates the weapon is
7                           // ready to fire.
8      int num_weapons;
9  } Spaceship;
10
11 // YOUR TASK: Implement charge_weapons and attack_ship.
12 void charge_weapons(Spaceship* ship);
13 void attack_ship(Spaceship* from, Spaceship* to);
```

Structs

```
1 typedef struct {  
2     char* name;  
3     int health;  
4     int* weapon_statuses;  
5     int num_weapons;  
6 } Spaceship;
```

- A struct is simply a collection of one or more variables, grouped together under a single name.
- As defined, our `Spaceship` struct contains the variables `name`, `health`, `weapon_statuses`, and `num_weapons`.

Structs

C defines two operators for dealing with structs. A member of a struct is referred to with the `.` operator, as shown below:

```
1 Spaceship starship = ...  
2 printf("The name of our ship is %s", starship.name);
```

However as you can see from the function declarations in `structs.c`, dealing with a pointer to a struct instead of structs themselves is quite common. To refer to a member of a struct from a pointer, use the `->` operator.

```
1 Spaceship *ship = ...  
2 printf("The name of our ship is %s", ship->name);
```


Structs

```
1  typedef struct {
2      char* name;
3      int health;
4      int* weapon_statuses; // An array whose elements indicate the
5                           // value of each weapon.
6                           // A value >= 100 indicates the weapon is
7                           // ready to fire.
8      int num_weapons;
9  } Spaceship;
10
11 // YOUR TASK: Implement charge_weapons and attack_ship.
12 void charge_weapons(Spaceship* ship);
13 void attack_ship(Spaceship* from, Spaceship* to);
```

- Implement the two functions and test your program.
- Note that to `weapon_statuses` is an array, so to access an element you apply an index like so:
`ship->weapon_statuses[0]`.

Headers

Now `cd` into the `task7` directory and open `headers.c` in your text editor.

You'll find an incomplete implementation of a linked-list struct.

```
1 // headers.c
2
3 // YOUR TASK: Write a header file for the linked-list
4 // implementation below, exposing list_append(), list_prepend(),
5 // and the struct itself.
6
7 #include "headers.h"
8
9 struct List {
10     void* data;
11     struct List* next;
12 };
13
14 ...
```

Headers

- Think of header files as the public documentation of any program or library you write.
- I.e., declarations for any functions we would consider “public” in an OOP language go in our header file. Declarations of functions we would consider “private” can remain in our source file.
- Anyone using the provided linked-list library would probably want the ability to declare instances of `struct List` and append and prepend to it, so we declare those in the header.

Further Work

If you'd like to continue your studies, please consider:

- Procuring a copy of *The C Programming Language* by Kernighan and Ritchie.
- Visiting the C tutorials on Lynda.com, free to all students at CSUCI.
- Familiarizing yourself with the command line through online courses such as those at Codecademy.com.