Computer Programming: Skills & Concepts (CP)
Functions

Julian Bradfield

Monday 9 October 2017

Summary of Lecture 6

- The `while` statement.
- The `for` statement.
- `fibonnaci.c`

This Lecture: Functions

- Motivation: reading dates
- Functions formally
- Scoping
- (Time allowing) live coding

Read a day and a month.

Day Between 1 and 31
Month Between 1 and 12

Months have different numbers of days, but we'll ignore that.

/* Read a Day */

```
int day = 0;
while (day < 1 || day > 31) { /* Ask until day is valid */
  printf("Enter a value between 1 and 31:");
  scanf("%d", &day);
}
```

```c
/* Read a Day */
int day = 0;
while (day < 1 || day > 31) { /* Ask until day is valid */
  printf("Enter a value between 1 and 31:");
  scanf("%d", &day);
}
/* Read a Month */
int month = 0;
while (month < 1 || month > 12) {
  printf("Enter a value between 1 and 12:");
  scanf("%d", &month);
}
```

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  /* Read a Day */
  int day = 0;
  while (day < 1 || day > 31) { /* Ask until day is valid */
    printf("Enter a value between 1 and 31:");
    scanf("%d", &day);
  }
  /* Read a Month */
  int month = 0;
  while (month < 1 || month > 12) {
    printf("Enter a value between 1 and 12:");
    scanf("%d", &month);
  }
  printf("You entered %d of %d\n", day, month);
  return EXIT_SUCCESS;
}
```

```c
/* Read a Day */
int day = 0;
while (day < 1 || day > 31) { /* Ask until day is valid */
  printf("Enter a value between 1 and 31:");
  scanf("%d", &day);
}
/* Read a Month */
int month = 0;
while (month < 1 || month > 12) {
  printf("Enter a value between 1 and 12:");
  scanf("%d", &month);
}
```

These are very similar. Write it once?

Generic: Read a value from 1 to maximum

```
/* To Do: Get maximum from somewhere. */
int value = 0;
while (value < 1 || value > maximum) {
  printf("Enter a value between 1 and %d:", maximum);
  scanf("%d", &value);
}
/* Use value for the day or month. */
```

Function: Read a value from 1 to maximum


```
int ReadValue(int maximum) {
  int value = 0;
  while (value < 1 || value > maximum) {
    printf("Enter a value between 1 and %d:", maximum);
    scanf("%d", &value);
  }
  /* Use value for the day or month. */
  return value;
}
```

Functions 'wrap up' a piece of code.

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

int ReadValue(int maximum) {
  int value = 0;
  while (value < 1 || value > maximum) {
    printf("Enter a value between 1 and %d:", maximum);
    scanf("%d", &value);
  }
  return value;
}
int main() {
  int day = ReadValue(31);   /* Call with maximum = 31 */
  int month = ReadValue(12); /* Call with maximum = 12 */
  printf("You entered %d of %d\n", day, month);
  return EXIT_SUCCESS;
}
```

Functions Formally

A function encapsulates code.

```
int ReadValue(int maximum) { ... }
```

Like a mathematical function, it takes inputs and returns a value.
Unlike a mathematical function, it can return different values:

```
/* Ask the user then print the value */
printf("%d", ReadValue(12));
/* Ask the user again then print the new value */
printf("%d", ReadValue(12));
```

Always Have a Return Statement

Bad Code:

```
int AddTwo(int to) {
  to + 2;
}
```

Good Code:

```
int AddTwo(int to) {
  return to + 2;
}
```

Returning Nothing

`void` means there is nothing to return. `return` can be omitted for `void`.

Bad Code:

```
int SayHello() {
  printf("Hello\n");
}
```

Good Code:

```
void SayHello() {
  printf("Hello\n");
  return;
}
```

Good Code:

```
void SayHello() {
  printf("Hello\n");
}
```

Multiple Inputs

Functions can have any number of arguments (inputs).

```
int Sum0() { return 0; }
int Sum1(int first) { return first; }
int Sum2(int first, int second) { return first + second; }
```

Return Happens Immediately

Bad Code:

```
/* Actually returns value */
int AddTwo(int value) {
  return value;
  /* Never reached */
  value = value + 2;
}
```

Good Code:

```
int AddTwo(int value) {
  value = value + 2;
  return value;
}
```

Returning Early

`return`'s immediate effect can be useful:

```
/* Take the absolute value of the number */
int AbsoluteValue(int number) {
  if (number >= 0) {
    /* All non-negative numbers will return here. */
    return number;
  }
  /* Now we know that number < 0 */
  return -number;
}
```

Scoping

Functions do not share variables.

Bad Code:

```
int AddTwo() { return value + 2; }
void Print() {
  int value = 0;
  printf("%d", AddTwo());
}
```

Scoping

Functions do not share variables.

Bad Code:

```
int AddTwo() { return value + 2; }
void Print() {
  int value = 0;
  printf("%d", AddTwo());
}
```

Good Code:

```
int AddTwo(int value) { return value + 2; }
void Print() {
  int value = 0;
  printf("%d", AddTwo(value));
}
```

Names Change

Good Code:

```
void PrintTwo(int start) {
  printf("%d\n", start + 2);
}
int main() {
  int alice = 0, bob = 1;
  PrintTwo(alice); /* Run PrintTwo with start = 0 */
  PrintTwo(bob);   /* Run PrintTwo with start = 1 */
  return EXIT_SUCCESS;
}
```

Value of `alice` or `bob` becomes `start`

Formalizing Function Calls

Functions run in an *environment*.
When we call a function:

1. Create a new environment.
2. Copy the arguments to their new names.
3. Run the function.
4. Save the return value.
5. Clean up the environment.

Formalizing Function Calls

Functions run in an *environment*.
When we call a function:

1. Create a new environment.
2. Copy the arguments to their new names.
3. Run the function.
4. Save the return value.
5. Clean up the environment.

This is why functions can't see variables outside their environment.

Call Example

```
int AddTwo(int value) {
  return value + 2;
}
int main() {
  int alice = 0;
  int bob = AddTwo(alice);
  printf("%d", bob);
  return EXIT_SUCCESS;
}
```

1. Make an environment for `AddTwo`.
2. Copy `0` to `value` (since `alice` is `0`).
3. Run `value + 2`.
4. Save the `return` value.
5. Restore `main`'s environment.
6. Continue running `main`, which will copy `return` to `bob`.

Arguments Are Copies

Changes do not impact callers:

```
void FailsToIncrement(int value) {
  /* Increments a copy in its own environment */
  value = value + 1;
}
void Fail() {
  int number = 0;
  /* Send a copy of number */
  FailsToIncrement(number);
  /* Prints 0 */
  printf("%d", number);
}
```

Declare Before Use

Bad Code:

```
int AddFour(int to) {
  /* AddTwo not yet declared */
  return AddTwo(AddTwo(to));
}
int AddTwo(int to) {
  return to + 2;
}
```

Declare Before Use

Bad Code:

```
int AddFour(int to) {
  /* AddTwo not yet declared */
  return AddTwo(AddTwo(to));
}
int AddTwo(int to) {
  return to + 2;
}
```

Good Code:

```
int AddTwo(int to) {
  return to + 2;
}
int AddFour(int to) {
  return AddTwo(AddTwo(to));
}
```

Declaring Separately

Good Code:

```
/* Tell the compiler that AddTwo is a function taking an int
   and returning an int.  No need to name variables. */
int AddTwo(int);

int AddFour(int to) {
  return AddTwo(AddTwo(to));
}
int AddTwo(int to) {
  return to + 2;
}
```

Summarizing Functions

Functions let you encapsulate code.

Use functions to:

- Reduce repetition
- Group functionality together
- Explain what a chunk of code does (by naming it)

Learning more

Reading Assignment:
Chapter 5 (sections 5.1-5.5) of "A book on C".

Lecture 8:
We will continue the "Functions" theme in Lecture 8.