Computer Programming: Skills & Concepts (CP)
Functions and Pointers

Julian Bradfield

Tuesday 10 October 2017

Last time:

- Functions
- return

This time:

- Global variables
- Motivation for Pointers
- Addresses aka Pointers

Aside: You can mix types

Good Code:

```
float Round(double numerator, int decimal_places);
```

Global Variables

```c
/* Declare a global variable. */
/* Notice this is outside a function. */
int i;

void print_i() {
  /* i is accessible from any function */
  printf("%d", i);
}

int main() {
  /* i is accessible from any function */
  i = 1;
  print_i();
  return EXIT_SUCCESS;
}
```

Global Variables Are Bad

```
int day;

int GetMonth() {

...
}

int main() {
  day = 1;
  GetMonth();
  /* What does this print? */
  printf("%d", day);
  return EXIT_SUCCESS;
}
```

Global Variables Are Bad

```c
int day;

int GetMonth() {
  int month;
  printf("Enter a day and month:");
  scanf("%d %d", &day, &month);
  return month;
}

int main() {
  day = 1;
  GetMonth();
  /* What does this print? */
  printf("%d", day);
  return EXIT_SUCCESS;
}
```

Global Variables Are Evil

- Global variables can be read/written from any system module
  - In contrast, local variables only seen from a particular software module
- Excessive use of globals tends to compromise modularity
  - Changes to code in one place affect other parts of code via the globals
  - Think of it as **data flow spaghetti**

1973 February

GLOBAL VARIABLE CONSIDERED HARMFUL

W. Wulf, Mary Shaw
Carnegie-Mellon University

• •

The problems of indiscriminant access and vulnerability are complementary: the former reflects the fact that the declaror has no control over who uses his variables; the latter reflects the fact that the program itself has no control over which variables it operates on. Both problems force upon the programmer the need for a detailed global knowledge of the program which is not consistent with his human limitations.

[Wulf 1973, pp. 28,32]

© Copyright 2014, Philip Koopman. CC Attribution 4.0 International license.

Global Variables Are Evil

- Global variables can be read/written from any system module
  - In contrast, local variables only seen from a particular software module
- Excessive use of globals tends to compromise modularity
  - Changes to code in one place affect other parts of code via the globals
  - Think of it as **data flow spaghetti**

1973 February

GLOBAL VARIABLE CONSIDERED HARMFUL

W. Wulf, Mary Shaw
Carnegie-Mellon University

• •

The problems of indiscriminant access and vulnerability are complementary: the former reflects the fact that the declaror has no control over who uses his variables; the latter reflects the fact that the program itself has no control over which variables it operates on. Both problems force upon the programmer the need for a detailed global knowledge of the program which is not consistent with his human limitations.

[Wulf 1973, pp. 28,32]

© Copyright 2014, Philip Koopman. CC Attribution 4.0 International license.

11,528 global variables make Toyota cars unsafe

. . . what alternatives are there?

ReadDate function

```
int ReadDate() {
  int day = ReadValue(31);
  int month = ReadValue(12);
  return /* Problem: we can't return both day and month. */;
}
```

Problem

We can't return two ints.

ReadDate: try 2

Bad Code:

```
void ReadDate(int day, int month) {
  day = ReadValue(31);
  month = ReadValue(12);
}
```

Remember: arguments are copies. They won't impact the caller.

Addresses are one way around this...

Addresses (also known as Pointers)

Computers keep variables at numbered addresses:

(Photo: Stacalusa, CC-0 license)

Idea: tell `ReadDate` to put `day` in box 0211 and `month` in 0224.

Address of a Variable: &

```c
int main() {
  int i;
  /* Print the address of i (the number on its box) */
  printf("%p\n", &i);
  return EXIT_SUCCESS;
}
```

New notation:

`&i`	Address of `i`
`%p`	Formatting for pointers aka addresses

Addresses can be Stored

```
int i;
/* This stores the address of i */
int* address_of_i = &i;
/* Print the same value (the address of i) twice: */
printf("%p\n", &i);
printf("%p\n", address_of_i);
```

Notation:

`&i`	Address of `i`
`%p`	Formatting for addresses
`int*`	Type of an address to an int

Address Types

`int*` means an address to an `int`.

`double*` means an address to a `double`.

etc.

Good Code:

```
int i;
int* address_of_i = &i;
```

Good Code:

```
double value;
double* address_of_value = &value;
```

Bad Code:

```
double value;
double address_of_value = &value; /* Missing asterisk */
```

Using Addresses: *

Use * to access a variable at an address.

```
int i = 2;
int* address_of_i = &i;
```

Now i and *address_of_i are *interchangeable* (aliases).

Using Addresses: *

Use * to access a variable at an address.

```
int i = 2;
int* address_of_i = &i;
```

Now i and *address_of_i are *interchangeable* (aliases).

```
/* Both print 2. */
printf("%d\n", *address_of_i);
printf("%d\n", i);
```

Using Addresses: *

Use * to access a variable at an address.

```
int i = 2;
int* address_of_i = &i;
```

Now `i` and `*address_of_i` are *interchangeable* (aliases).

```
/* Both print 2. */
printf("%d\n", *address_of_i);
printf("%d\n", i);


/* This is the same as i = 3. */
*address_of_i = 3;
/* Prints 3. */
printf("%d\n", i);
```

Another Example

```
int i = 2;
/* & takes the address of i. Then * goes there. */
*(&i) = 3;
/* prints 3 */
printf("%d\n", i);
```

Not terribly useful, but instructive.

Summarizing

Variables live in memory. Memory is like a bunch of post boxes.
$\Longrightarrow$ Every variable has a numbered address.

To get that address, we use `&`.
To access the value at an address, we use `*`.
We can remember addresses. `int*` stores an address to an `int`.

Summarizing

Variables live in memory. Memory is like a bunch of post boxes.
$\Longrightarrow$ Every variable has a numbered address.

To get that address, we use `&`.
To access the value at an address, we use `*`.
We can remember addresses. `int*` stores an address to an `int`.

Three uses of the `*` symbol:

`*address_to_i`	Access a variable at an address
`int*`	Type of an address
`i * j`	Multiply `i` by `j`

Summarizing

Variables live in memory. Memory is like a bunch of post boxes.
$\Longrightarrow$ Every variable has a numbered address.

To get that address, we use `&`.
To access the value at an address, we use `*`.
We can remember addresses. `int*` stores an address to an `int`.

Three uses of the `*` symbol:

`*address_to_i`	Access a variable at an address
`int*`	Type of an address
`i * j`	Multiply `i` by `j`

Puzzle:

```
int puzzle(int j) {
  int* i = &j;
  return j**i;
}
```

What does `puzzle(3)` return?

Summarizing

Variables live in memory. Memory is like a bunch of post boxes.
$\Longrightarrow$ Every variable has a numbered address.

To get that address, we use `&`.
To access the value at an address, we use `*`.
We can remember addresses. `int*` stores an address to an `int`.

Three uses of the `*` symbol:

`*address_to_i`	Access a variable at an address
`int*`	Type of an address
`i * j`	Multiply `i` by `j`

Puzzle:

```
int puzzle(int j) {
  int* i = &j;
  return j * (*i);
}
```

What does `puzzle(3)` return?

Useful for Multiple Values

Good Code:

```
void ReadDate(int* address_of_day, int* address_of_month) {
  *address_of_day = ReadValue(31);
  *address_of_month = ReadValue(12);
}
int main() {
  int day, month;
  ReadDate(&day, &month);
  printf("You entered %d of %d", day, month);
  return EXIT_SUCCESS;
}
```

Useful for Multiple Values

Good Code:

```
void ReadDate(int* address_of_day, int* address_of_month) {
  *address_of_day = ReadValue(31);
  *address_of_month = ReadValue(12);
}
int main() {
  int day, month;
  ReadDate(&day, &month);
  printf("You entered %d of %d", day, month);
  return EXIT_SUCCESS;
}
```

Question: Aren't Arguments Copied?

Useful for Multiple Values

Good Code:

```
void ReadDate(int* address_of_day, int* address_of_month) {
  *address_of_day = ReadValue(31);
  *address_of_month = ReadValue(12);
}
int main() {
  int day, month;
  ReadDate(&day, &month);
  printf("You entered %d of %d", day, month);
  return EXIT_SUCCESS;
}
```

Question: Aren't Arguments Copied?

Answer: yes, the *addresses* are copied.

Addresses: Reach Into Another Environment

The program has one giant set of post boxes.
A function can access any of them. . . but needs the address.

Dangling Addresses

Bad Code:

```
int *Dangerous() {
  int i;
  return &i;
}
```

Remember from Lecture 7:
When a function returns, its environment is destroyed, *including* i.

Told the postman the box is unused, but somebody still has the address.

Summary: Escaping the Environment

Global Variables

Easy to use initially

Hard to know what a function does:

```
void ReadDate();
```

Addresses

Requires thinking about postboxes.

Explicitly documents what a function can do:

```
void ReadDate(int* day, int* month);
```

Following Up

For Functions in general:
'A Book on C', Sections 5.1-5.6
(please ignore the comments on 'traditional C' and C++)

For pointers:
'A Book on C', Sections 6.1-6.3