Pointers, Arrays and “Call by Reference”

CMSC 104 Section 02
May 1, 2024

Administrative Notes

Remember Quiz 5 is now next Wednesday, May 8

- There will be a sample quiz; we'll go over it in class next Monday

Remember that the final is Monday, May 20, 6-8 pm in this class

- If there's an issue about that time you need to let me know ASAP
- There will be "sample exam" we'll talk about in class on May 13

New variable types - “pointers”

A pointer variable in C is a variable that contains a memory address

- It does not contain an int, a float,...
- It contains an address - specifically, the address of a variable of some other type
- You have to tell the C compiler what type of variable will be found at that location in memory

“*” means ‘this is a pointer variable’

You tell C that a variable is a pointer by using the * symbol

```
int *quarters; //quarters is a pointer variable that points to the location of an integer

float *price; //price is a pointer variable that points to the location of a float

char *name; //name is a pointer variable that points to the location of a char or string
```

Stay with me on this; we’ll explain

A note on syntax: you have to put a * next to each pointer variable when you declare it

```
int *quarters, dimes;
```

quarters is a pointer; dimes is an integer

```
int *quarters, *dimes;
```

quarters is a pointer; dimes is a pointer

So what does this look like?

```
#include <stdio.h>

int main() {

  int *number;
  float *price;
  char *name;

  printf ("The values are: \n  number: %p \n  price: %p \n name: %p \n", number,
price, name);

  return 0;
}
```

```
 The values are:
   number: 0x7ffc4ade9248
   price: (nil)
  sale: 0x7f095b7398c0
  name: (nil)
```

We didn’t initialize these variables - assign meaningful values to them. They contain whatever random bits were left from the last time that memory was used. This “nil” is simply random ‘junk’.

So what does this have to do with arrays?

A variable that is the name of an array is really just a pointer

- It contains the address of the first element of the array
- If you print the array, you get an address

```
#include <stdio.h>
int main() {
  int numbers[4] = {1,2,3,4};
  int i;

  for (i = 0; i < 4; i++) {
    printf("%d \n", numbers[i]);
  }
  printf("%p \n", numbers);
  return 0; }
```

The results:

```
[arsenaul@linux1 ~/cs104]$ gcc
makechange.c -o makechange.o
[arsenaul@linux1 ~/cs104]$ ./makechange.o
1
2
3
4
0x7ffd9762d4b0
```

An array name is just a pointer to the array!!!

Remember that a string is just an array of characters

- It can be of definite or indefinite length
- Definite length: define the size as an array:

```
#include <stdio.h>

int main()  {

  char name[20] = "Wes Moore";
  char *surname = "Jones";

  printf("name is %20s \n surname is %s", name, surname);

  return 0;
}
```

```
[arsenaul@linux1 ~/cs104]$
./a.out
name is            Wes Moore
 surname is Jones
```

The array printed 20 characters, right-justified
The pointer printed the exact length

Strings are null-terminated

“\0” is the C representation of a null character

All strings end with \0 - that’s how C tells the end of a string

C will put that at the end of string for you if you don’t put it there yourself

```
char name[3] = {“A”, “l”, “\0”};  //you inserted the null

char name[3] = “Al” // C will insert the null
```

From Monday’s lecture

At the end of Monday’s lecture I said that “weird things happen if the parameters are arrays”

And I told you we’d cover that today. So here we go:

- Spoiler alert: this is how you implement “call by reference” in C.

“Call by reference”

C always passes parameters to a function using “call by value”

- The value of the actual parameter is copied from its location in memory to the separate location in memory associated with the formal parameter
- But - what if the actual parameter is a pointer variable?
  - The value is an address.
  - The address is copied
  - And the function winds up operating on the same actual locations in memory as the main program

This is referred to as “call by reference” - rather than pass a copy of the value, we pass a pointer to the actual location in memory

- The function can then directly impact the variables in the main program WITHOUT passing a return value

A quick note on Classwork 9 part 2

The goal of the “makechange” algorithm is presumed to be to make the necessary change using the smallest total number of coins. To do that:

- Give as many quarters as you can; you’ll have less than 25 cents to go
- Then give as many dimes as you can; you’ll have less than 10 cents to go when you’re finished
- Then give as many nickels as you can
- Finish it off with 0 through 4 pennies

If you need to give someone 93 cents, you could always just give that person 93 pennies. But that would likely not make people happy.

Part 2 of Classwork 9

```
#include <stdio.h>

int main () {
  int change[4];
  int cents;
  void makechange(int cents, int change[4]);


  printf("This program will figure out the change for you . \n\n");
  printf("Enter the number of cents: ");
  scanf("%d", &cents);

  makechange( cents, change);  //notice we're passing an array, which means
we're passing a pointer

  printf("Change for %d cents is: %d quarters, %d dimes, %d nickels and %d
pennies \n", cents, change[0], change[1], change[2], change[3]);


  return 0;
}
```

```
void makechange( int money, int coins[4]) {
  coins[0] = money/25;
  money -= coins[0] * 25;
  coins[1] = money/10;
  money -= coins[1] * 10;
  coins[2] = money/5;
  money -=  coins[2]* 5;
  coins[3] = money;
}
```

An alternative “makechange” routine

```
void makechange( int money, int coins[4]) {
  int i;
  int values[4] = {25, 10, 5, 1};
  for (i = 0; i < 4; i++) {
     coins[i] = money/values[i];
     money = money%values[i];
}
```

The program with some debugging statements

```
#include <stdio.h>

int main () {
  int change[4];
  int cents;
  void makechange(int cents, int change[4]);


  printf("THis program will figure out the change for you . \n\n");
  printf("Enter the number of cents: ");
  scanf("%d", &cents);

  printf(" Location of change: %p \n\n", change);


  makechange( cents, change);  //notice we're passing an array, which means we're
passing a pointer

  printf("Change for %d cents is: %d quarters, %d dimes, %d nickels and %d pennies
\n", cents, change[0], change[1], change[2], change[3]);


  return 0;
}
```

```
void makechange( int money, int coins[4]) {


  printf ("Location of coins: %p \n\n", coins);
  int i;
  int values[4] = {25, 10, 5, 1};
  for (i = 0; i < 4; i++) {
    coins[i] = money/values[i];
    money = money%values[i];
  }

}
```

Results of that program

```
This program will figure out the change for
you .

Enter the number of cents: 89
 Location of change: 0x7fff62832390

Location of coins: 0x7fff62832390

Change for 89 cents is: 3 quarters, 1 dimes,
0 nickels and 4 pennies
```

Notice that the array “change” (the actual parameter) and the array “coins” (the formal parameter) represent the exact same location in memory

So when the function changes the value of “coins” it ***ALSO*** changes the value of “change” in the main program.

So, what do we take from these two lectures?

1. Arrays are useful because they store multiple values at once
   a. This lets you perform a lot of calculations more easily, and a perform a lot of calculations that you otherwise couldn’t
2. Pointers are variables that contain addresses of other variables
   a. You indicate that a variable is a pointer with *
   b. You have to know what type is stored in the variable whose address you have
   c. Print a pointer variable using “%p”
3. Pointers and arrays are the same thing - or two sides of the same coin
   a. An array name is a pointer to the array

4. You can implement “call by reference” functions in C using pointers (or arrays, since they’re the same)

   a. C copies the value from the main program symbol table to the function symbol table, but what is copied is the address to use

5. A string is an array of char’s, it’s also a pointer to a char (array)

   a. Strings are null terminated - \0 indicates the end of the string