CS 201
Pointers (2)

Debzani Deb

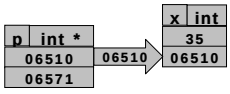
Overview

- Pointers
- Functions: pass by reference
- Quiz 2 : Review
- Q & A

Uses of & and *

p = &x;

// p points at x now.



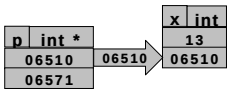
y = * p;

// y has the value pointed out by the pointer p.

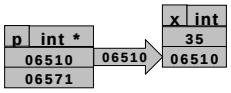
* p= 13;

// 13 was inserted to the

// place pointed out by p.



Cont.



x	&x	p	*p	&p
35	06510	06510	35	06571

- & when applied to a variable, yields its address (pointer to the variable).
- * when applied to an address (pointer), fetches the value stored at that address.

Arithmetic and Logical Operations on Pointers.

- A pointer may be incremented or decremented.
- An integer may be added to or subtracted from a pointer.
- Pointer variables can be used in comparison, but usually only in a comparison to NULL.

Arithmetic Operations on Pointers

- When an integer is added to a pointer, the new pointer value is changed by the integer times the number of bytes in the data variable the pointer is pointing to.
- Example

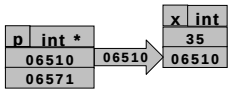
p = &x;

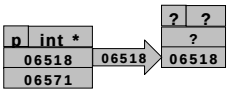
// size of int is 4 bytes

p = p + 2;

// address is increased

by 8 (2*4) bytes.





What is the use of Pointers?

- Pointers can be used to operate on variable-length arrays.
- Pointers can be used for advanced data structures.
- Pointers can be "cheaper" to pass around a program.
- You could program without using them, but you would be making life more easier by using them.

The true horror of pointer

- Each pointer Always points something.
- No bounds checking – pointers can point outside the array, even outside the program.
- No type checking – you can cast a pointer to anything.
- Memory Leaks – you can forget to deallocate storage when you're done with it.
- Dangling references – You can deallocate storage before you're doe with it.
- **You just have to be careful while using pointers.**

Conclusions

- Pointers allow more sophisticated programming.
- But if you mess up with pointers, the things will be real messed up.
- It is best to only use pointers once you are confident with simpler programming techniques.
- Pointers bugs are the hardest to find – you might find your program crashes randomly at different points in the code. This is typical of pointer bugs.

Functions: Pass by reference

Functions

- In Chapter 3 We looked at 4 types of functions
  - No arguments, no return value
    - void printData (void);
  - One or more arguments, no return value
    - void drawCircle (int radius);
  - One or more arguments, one return value
    - double square (double num);
- What if we need to return more than one value from a function?

Functions: Arguments are passed by values

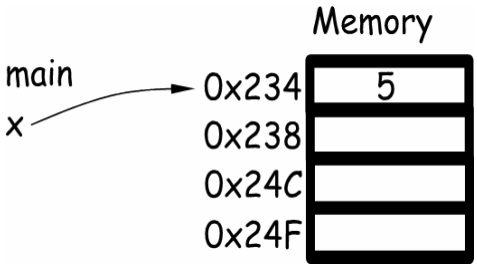
- **Argument lists** are used to communicate information from the main function to its function subprograms.
  - Arguments make functions more versatile because they allow us to execute the same function with different sets of data.
- **Return values** are used to communicate information from the function subprogram back to the main program.
- We can use **output parameters** to return multiple results from a function.
- When a function is called, it is given a **copy of the values** that are passed in as arguments.
  - If you manipulate the value of an argument, it has no impact on its value in the main function
  - Therefore, these are called **input parameters**, because they can only bring information into the function, and not back out.

Example with pass by value

```
void myFunc(int arg);

int main(void)
{
  int x = 5;
  myFunc(x);
  printf("%d\n", x);
}
void myFunc(int arg)
{
  arg = 4;
}
```

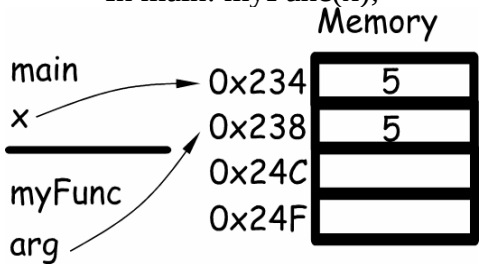
```
/* Output */
5
```


In main: int x = 5;



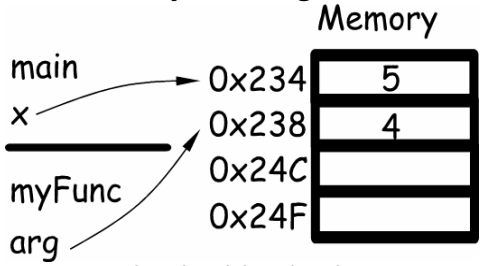
In main, **x** is assigned the value 5.
This places the value 5 in the memory cell reserved for x
In this case, it is at address 0x234

In main: myFunc(x);



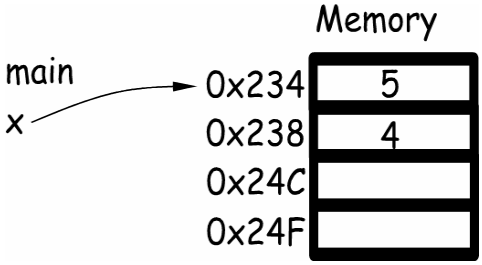
We call the function myFunc and pass it the value of x
myFunc allocates a new memory cell for its formal parameter arg
The value 5 (a **copy** of the value in x) is placed in arg

In myFunc: arg = 4;



In myFunc, **arg** is assigned the value of 4
This places the value 4 in the memory cell for **arg**
This is not the same cell as ***x***

In main: printf("%d\n", x);



Back in main, when we print out **x**, the value it points to is still 5.

Functions: Arguments are passed by Reference

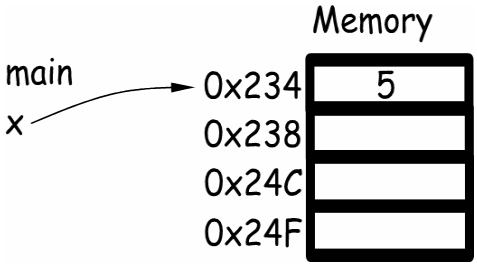
- What if we want our changes to the value in the function to affect the value in the main function?
- We can accomplish that by passing the address of a variable as argument to a function and manipulate that variable inside the function.
- In the formal parameter list, we put a * in front of the parameter name.
  - This defines a **pointer,** which means that we will be passing the **address** of the value, rather than the value itself.
- In the function call, we put an & in front of the argument name.
  - The & tells the compiler to pass the **address** of the variable, not its value.
- When we need to access the value of the argument in the function, we put a * in front of the variable name
  - This * tells the compiler to access the value pointed to by the address in the variable.

Example with pass by reference

```
void myFunc(int* arg);

int main(void)
{
  int x = 5;
  myFunc(&x);
  printf("%d\n", x);
}
void myFunc(int* arg)
{
  *arg = 4;
}
```

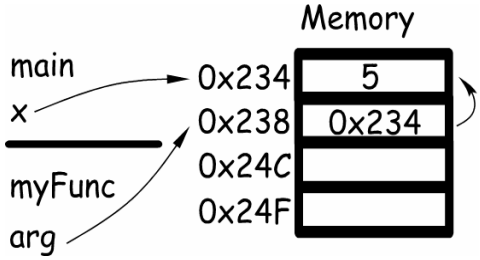
```
/* Output */
4
```


In main: int x = 5;



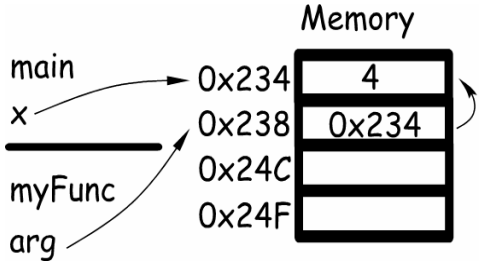
In main, **x** is assigned the value 5, just like before.
The address of the memory cell for x is 0x234
The value of &x is 0x234

In main: myFunc(&x);



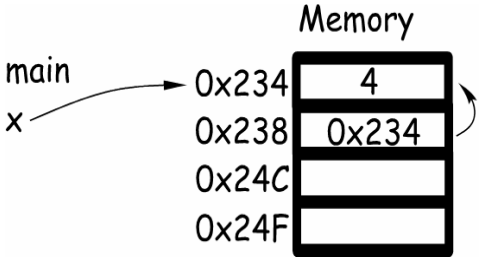
When we call myFunc, we pass it &x, the address of x
This value is stored in the memory cell for arg.
arg == 0x234, *arg == 5

In myFunc: *arg = 4;



When we set the value of *arg, we are setting the value **pointed to** by arg – the value at 0x234

In main: printf("%d\n", x);

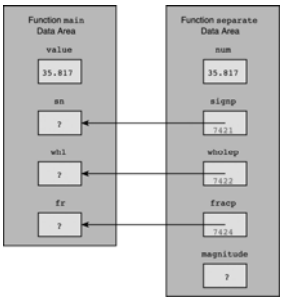


Back in main, the value of x is now 4


```
#include <stdio.h>
#include <math.h>

void separate(double num, char *signp, int *wholep,
double *fracp);

int main(void)
{
  double value, fr;
  char sn;
  int whl;

  printf("Enter a value to analyze> ");
  scanf("%lf", &value);

  separate(value, &sn, &whl, &fr);
  printf("Parts of %.4f\n sign: %c\n", value, sn);
  printf(" whole number magnitude: %d\n", whl);
  printf(" fractional part: %.4f\n", fr);
  return 0;
}
```

```
void separate(double num, char *signp, int *wholep,
double *fracp)
{
  double magnitude;
  if (num < 0)
    *signp = '-';
  else
    if (num == 0)
      *signp = ' ';
    else
      *signp = '+';
  magnitude = fabs(num);
  *wholep = floor(magnitude);
  *fracp = magnitude - *wholep;
}
Parts of 35.8170
sign: +
whole number magnitude: 35
fractional part: 0.8170
```


Figure 6.4 Parameter Correspondence for separate(value, &sn, &whl, &fr);

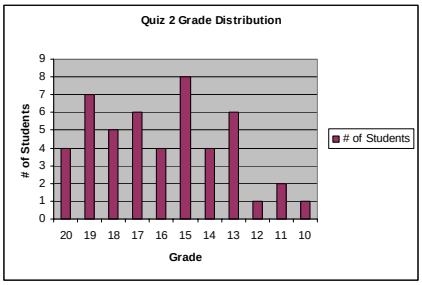


Scope of Names

- The scope of a name refers to the region of a program where a particular meaning of a name is visible or can be referenced.
- #define variables scope begins at their definition and ends at the end of the source file. All functions can "see" these variables.
- The scope of the name of a function begins with the function prototype and ends with the end of the source file.
- All formal parameter names and local variables are visible only from their declarations to the closing brace of the function in which they are declared.
- Look at Fig 6.8 and pay particular attention to Table 6.4 until you agree with ALL the visibilities.

Quiz 2: Review

Quiz 2 Grade Distribution



Quiz 2: Tricky Questions

- The following decision structure is **invalid**:

```
if x <= y
   printf("%1f", x);
else
   printf("%1f", y);
```

- True: gives you syntax error because the condition is not enclosed in braces.
- Correct use: `if (x<=y)`

Quiz 2: Tricky Questions

- The statements on the left always give p the same value as the code on the right, but the code on the right may execute faster.

```
if (x > 15)                    if (x > 15)
    p = p * x;                     p = p * x;
if (x > 30)                    else if (x > 30)
    p = 2 * p * x;                 p = 2 * p * x;
```

- False: Different values are assigned to p when x>30.

when p = 2 and x = 20

p is 40, 1st rule triggered — p is 40, 1st rule triggered

when p = 2 and x = 40

p is 6400, both rule triggered — p is 80, 1st rule triggered

Quiz 2: Tricky Questions

n || a <= b && c != 100

The complement is

!(n || a <= b && c != 100)

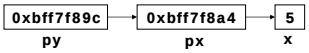
= !n && !((a <= b) && (c != 100))

= !n && (!(a <= b) || !(c != 100))

= !n && ((a > b) || (c == 100))

Questions asked during lecture

Questions about Pointer




```
int x;
int *px;
int **py;

px = &x;
py = &px;
*px = 5;
printf("X is %d\n",x);                  5
printf("*px is %d\n", *px);             5
printf("px is %p\n",px);                0xbff7f8a4
printf("&x is %p\n",&x);                0xbff7f8a4
printf("*(&x) is %d\n",*(&x));          5
printf("&px is %p\n",&px);              0xbff7f89c
printf("*py is %p\n", *py);             0xbff7f8a4
```

What is *py?

Why do you want us to use fabs in lab4?

- Because of the way double variables are stored (we will see more about this in chapter 7), not all numbers can be represented exactly. Therefore, **an equality comparison of two type double values can lead to surprising results.**
- What if the input is (0.00000001234, 0.0) as (x, y) coordinate value to your program?
  - There can be a various representation of the number 0.00000001234 depending on hardware, OS or even compiler. E.g. 0.000000, 0.000000012 etc.
  - You want your code to act consistently irrespective of the environment it is executing on.

Example of Challenging Questions

1. What will happen in the following cases. Explain very briefly (one or two lines)

   printf("%d %d\n", 1, 2, 3);

   printf("%d %d %d \n", 1, 2);
2. We have seen that all pointer variables (i.e. int *, double *) are of same size (in our case all of them occupy 4 bytes of memory and contain plain and simple addresses). Considering this, can we live without declaring the type of a pointer variable? If your answer is 'yes', then explain why? If your answer is 'no' then give at least one example showing the absolute necessity of the declaration of pointer variable.