C Programming

Pointers

Passing Arrays As Arguments

Intro To Pointers

What's a Pointer?

A variable that contains

an address

of a

location in memory

Can point to
different types,
depending on the declaration

"Depending on the Declaration"?

Has an asterisk between the data type and the name

The asterisk now becomes part of the data type

e.g.:

int *pNumber;

This points at an int.

What's With the "int"?

What we want to find

at the address

that we're looking at

int before the asterisk indicates that we'll find an *int* at that location.

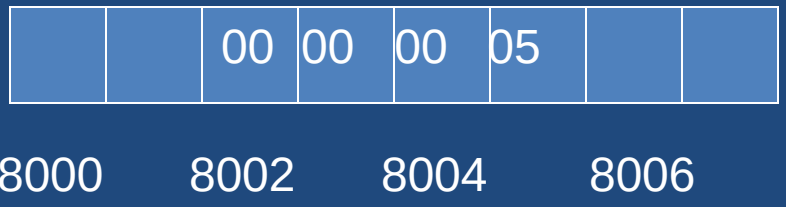
A Declaration Example

e.g. int * pNumber = 0x8002;

This indicates that we have a variable called pNumber that is a pointer to an int located at address 0x8002.

- 0x before a number indicates a hexadecimal number
- 0 before a number indicates an octal number

Further ...



If pNumber pointed to address 0x8002, it would find the 4 byte int located there.

Analogy

It's like
an address
on a map

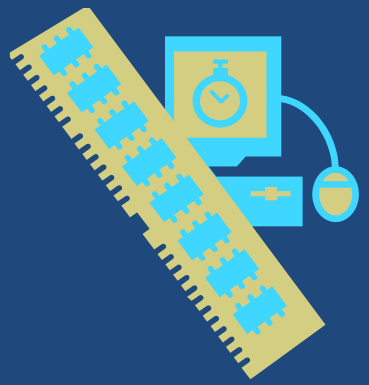
308 King St. W., Kitchener, ON

The land that the buildings are

on

are like

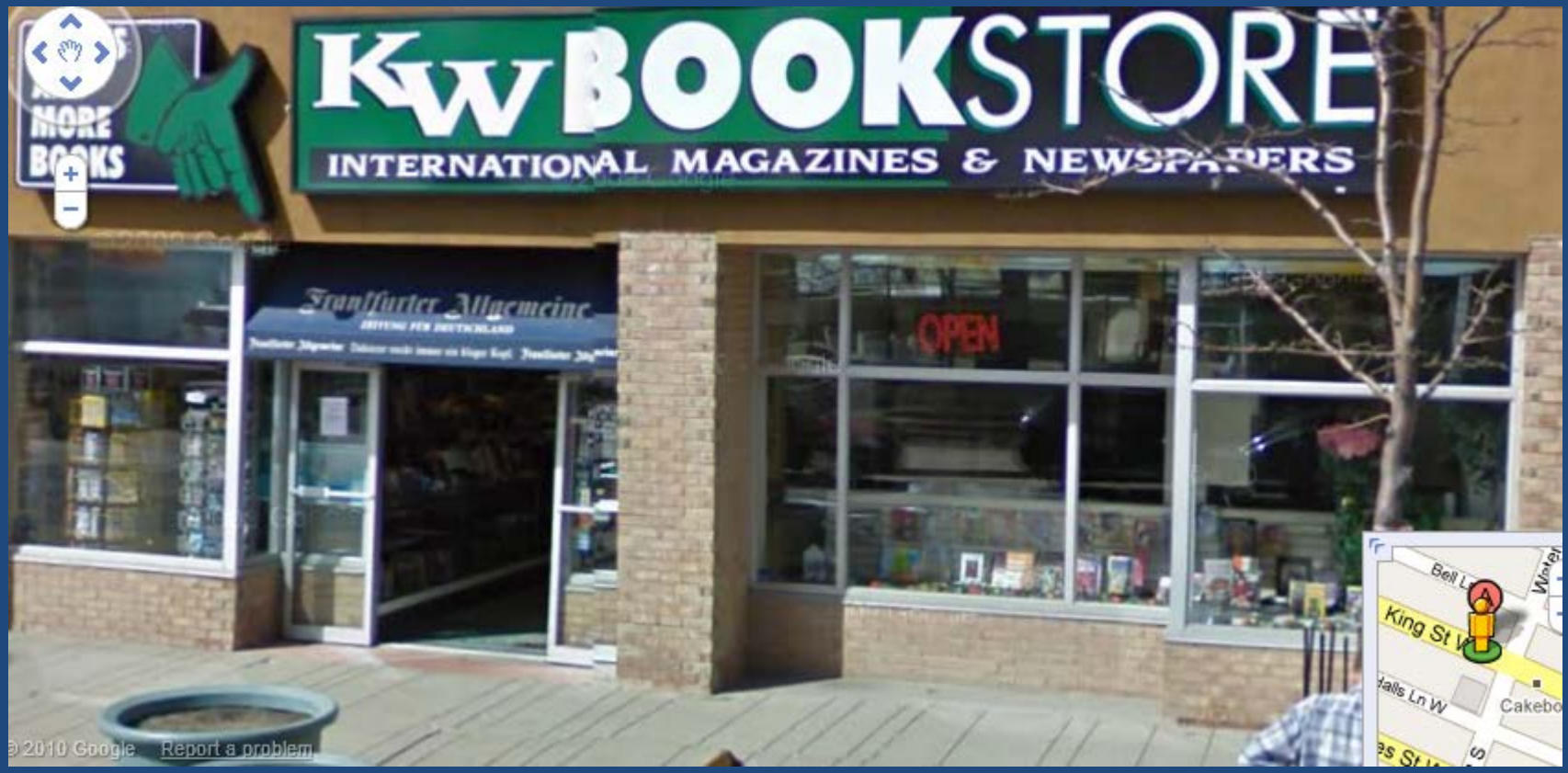
your computer's memory



You can build whatever you want on the land

You can store whatever you want in the memory

But, ultimately, there's only one meaningful thing at that address

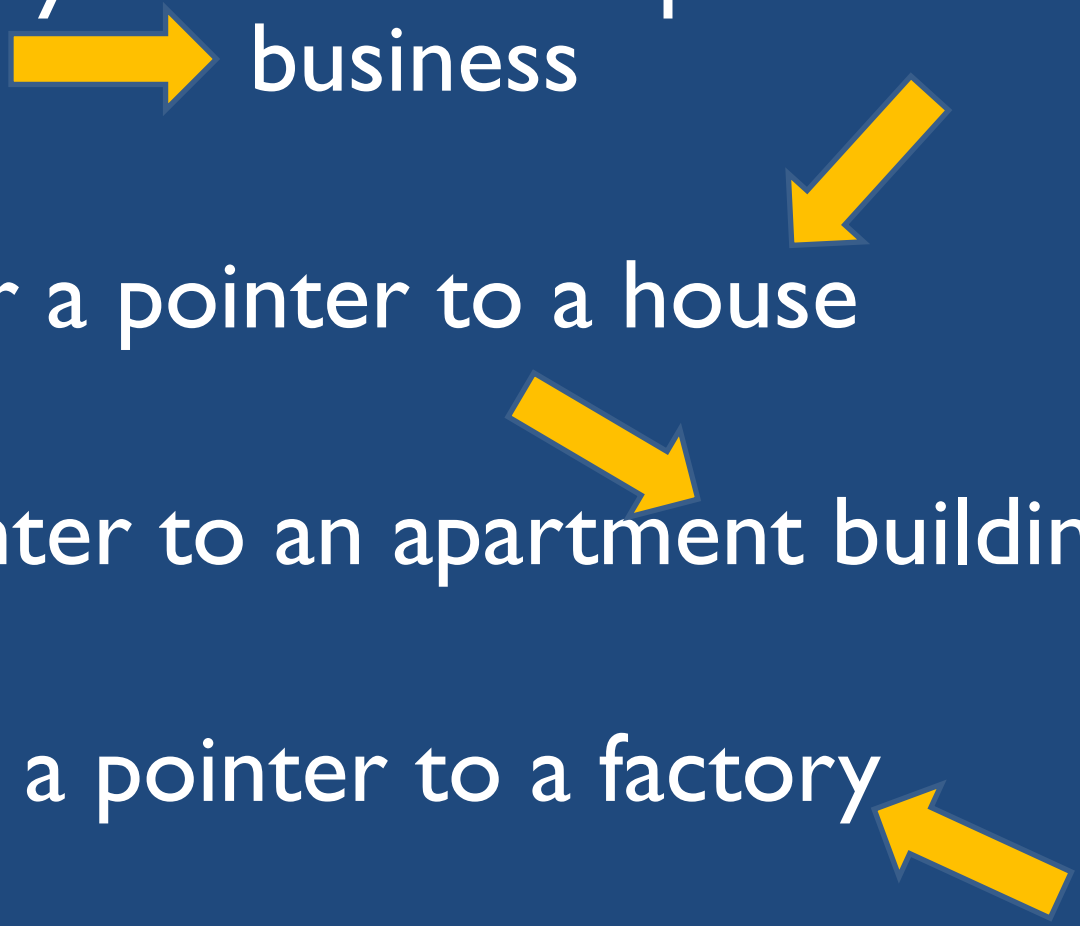

KW BOOKSTORE
INTERNATIONAL MAGAZINES & NEWSPAPERS
MORE BOOKS
Frankfurter Allgemeine
OPEN
King St W
Cakebo
© 2010 Google
Report a problem

You can say that there's a pointer to a business

Or a pointer to a house

Or a pointer to an apartment building

Or a pointer to a factory



So, if you wanted to find KW Bookstore, someone could give you the address of it

Important Initialization Note

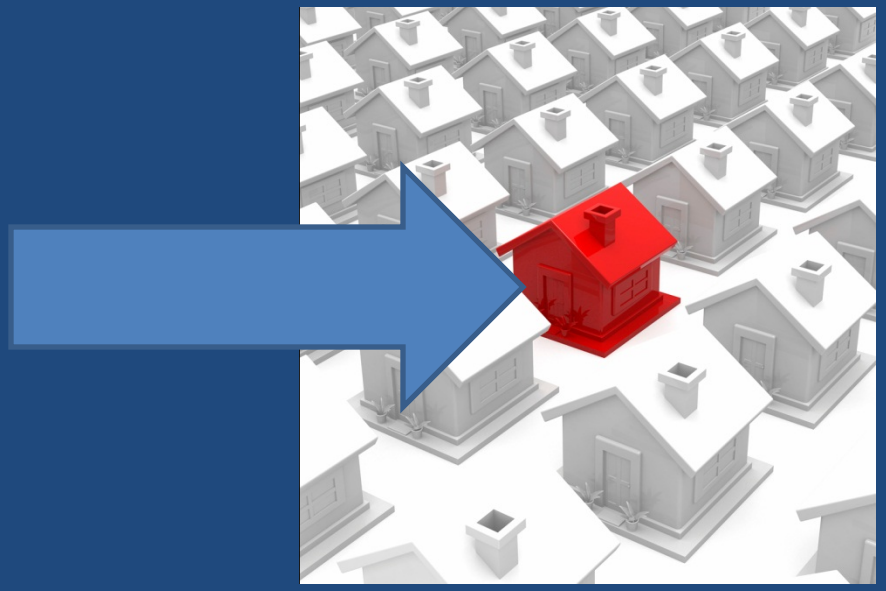
You **must** always initialize your pointer variables upon declaration.

You can use NULL, which is a constant that is defined in stdio.h

- e.g. int * pNumber = NULL;

OK, Now What?

We can use pointer variables to access what is at the addresses that we have set them to



If we want to do something with what is at that address, we dereference the pointer variable

De-what????

You dereference a pointer variable by putting an asterisk in front of the variable **when you use it.**

e.g. printf("%d\n", *pNumber);

Dereferencing

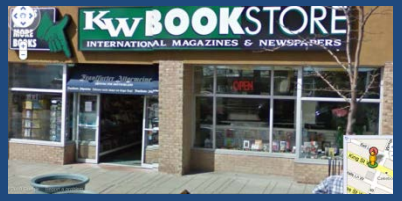
Dereferencing can be thought of as "go to the location that the pointer is pointing to and do something with what is there".

e.g. printf("%d\n", *pNumber);

"Go to the location that **pNumber** is pointing to and **get the int located there so we can send it to printf().**“

Why an int? Because we said that pNumber was a pointer to an int when we declared it.

Going back to the analogy, you can "go to the address of the business at 308 King St. W., Kitchener, ON and buy something"



Sooo...

So, this example will print the value 5 if it is running on a Freescale processor (which has the bytes of an int in reverse order from an Intel processor).

Changing a Value Being Pointed To

8000 8002 8004 8006

```
int  *pNumber = 0x8002;
*pNumber = 6;
```

OK, Now So What?

This, in itself, isn't overly useful.

It is a demonstration of how pointers work.

Usually, though, you don't set a pointer to point to a specific address.

- Usually, the pointer points to a variable.

Misc. Bit of Syntax

Putting an ampersand (&) in front of a variable gives the address of that variable.

More Realistic, Somewhat Useless Example

```
int  daysWorked = 60;
int * pNumber = &daysWorked;

   *pNumber = 40;
   printf("%d\n", *pNumber);        // prints 40
   printf("%d\n", daysWorked);  // prints 40
```

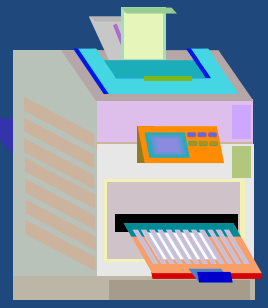
Indirection

This behaviour is called indirection, since the daysWorked variable is indirectly changed through dereferencing the pointer variable.

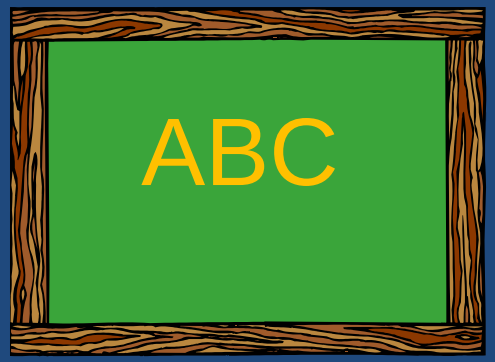
Review: Passing Arguments by Value

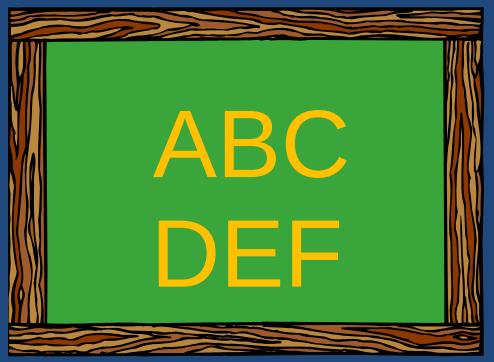
Remember before when we talked about passing arguments to a function.

If you pass an argument to a function, **a copy** of the argument gets passed to the function.



Thus, anything you change on the copy **only** gets changed on the copy, not on the original.





What if the Parameter is a Pointer?

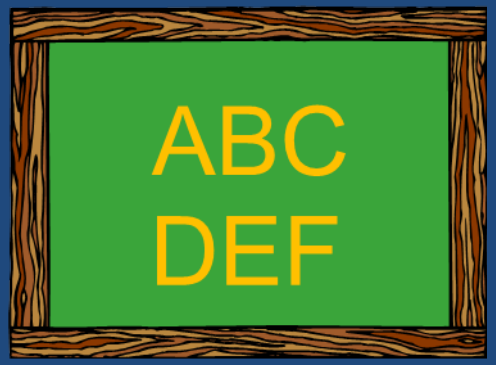
If the parameter is a pointer, an **address** gets passed.

You can **dereference the pointer.**

That means that you have complete access to whatever the pointer is pointing to.

Implication

If you **change the dereferenced pointer,** the **original** value is changed as well!!!



Useful Example, Part 1

```
void getPair(int *pNum1, int *pNum2);

int main(void)
{
int base = 0;
int exp = 0;

    getPair(&base, &exp);
    printf("%d %d\n", base, exp);
    return 0;
}
```

Example, Part 2

```
void getPair(int *pNum1, int
  *pNum2)
{
  *pNum1 = getNum();
  *pNum2 = getNum();
}
```

Explanation

We want getPair() to get two numbers from the user.

We can't return two numbers as return values, though.

What that will let us do is change as many arguments as we want within the function.

- All we have to do is pass the addresses of the variables we want to change!

Pass By Reference

This is called Pass By Reference.

Arrays

When passing an array as a parameter, the array is automatically passed by reference.

The reason is that **the name of an array is the same as the address of its very first element.**

Passing an Array

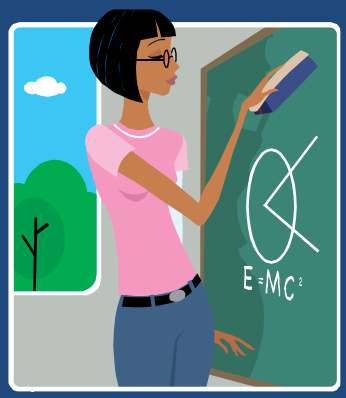
Thus, a function that takes an array as a parameter can change that array in the function and have the array values change in the calling function!

Array Example, Part 1

```
int main(void)
{
int inventory[12] = {3, 4, 5};
    printAndClearInv(inventory);
    return 0;
}
```

Array Example, Part 2

```
void printAndClearInv(int inv[])
{
int i = 0;
    while( i < 12 )
    {
        printf("%d\n", inv[i]);
        inv[i] = 0;
        i++;
    }
}
```



Syntax Note

You shouldn't specify the size of the array in the parameter list.

Style Note

It is common to start pointer names with the letter 'p'.

Making Passed Arrays Safe Against Change

By default, arrays are passed by reference, so they're totally accessible to the function to change

If you don't like that (and you wouldn't like it in many cases), you can keep the array safe by using const

e.g. void printInv(const float inv[]);

Any attempts to change the contents of inv will cause a compiler error

Pointer Arithmetic

Looking At Arrays Again

Recall: An array is a variable with many related elements.

Question: How is it stored?

Array Organization

The elements of an array are stored next to each other, starting with element 0.

- This is called "contiguously".

So, when you declare an array of ints, element 0 is first, then right next to it is element 1, then element 2, etc.

0	1	2	3	4	5	6	7	8
87	333	1993	3	-3	31	8	3	22

Pointers Accessing Arrays

You can set up a pointer to access elements of an array.

In doing this, you set the pointer to contain the address of the array.

- For arrays only, the name of the array is the same as the address of its very first element.

e.g.:

- int inventory[5] = {3, 4, 5, 6, 7};
- int *pNumber = inventory;

Changing Elements of the Array Using a Pointer

```
*pNumber  = 2;
 /* this changes the value of the int at that address (the start of the array) to have the value 2 */


printf("%d\n", inventory[0]); // prints 2
printf("%d\n", *pNumber); // prints 2
```

How About Changing the Address?

e.g. pNumber++;

- This changes the address that pNumber is pointing to, **not** the contents at that address.
- What's the difference? The absence of the asterisk.

One Slight Issue ...

The example used ++.

Ordinarily, that would mean that you'd be adding 1 to the value.

In the case of pointers, though, you're adding the size of one of the items being pointed to instead.

If we're pointing at an int, it adds the size of an int to the address.

So, if the address pointed to before was 8000, it's now pointing at 8004 (because our int is 4 bytes in size).

Pointer Arithmetic

This behaviour is called Pointer Arithmetic.

In addition, there are some arithmetic operations that we can't do with pointers:

- multiplication and division
- subtracting a pointer from a number
- adding two pointers

Arithmetically, What Can We Do To Pointers?

We can:

- increment and decrement
- add integer to a pointer
- subtract an integer from a pointer (but not the other way around!)
- subtract one pointer from another

Why?

These operations make sense; the others don't.

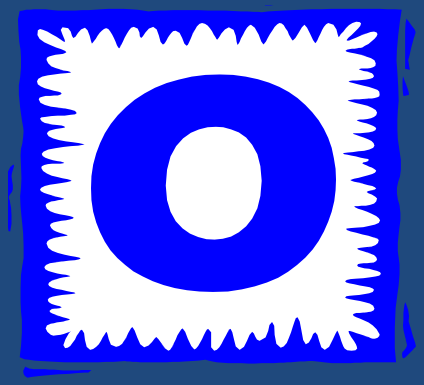
Important Point To Remember About Pointers

Any arithmetic done on a pointer is done based on **the size of the item to which it is pointing.**

Implication

That means that anything we can do to array elements, we can do using a pointer to the start of the array and pointer arithmetic.

Aside: This is why array indices start at 0. It is analogous to adding 0 to a pointer that is pointing to the start of the array.



sizeof

sizeof

The sizeof operator can be used to determine how many bytes a particular variable or data type is

e.g. sizeof (int) gives the number of bytes taken up by an int

Caveat!

This does NOT work with arrays that are passed as parameters, since the name of an array is treated as a pointer variable.

To find the size of an array passed as a parameter, take the number of elements (from the constant that you created!) and multiply it by the size of the item contained in the array.