C Programming

Scope and Storage Classes

Style

Scope and Storage Classes

Where Are Variables Available?

So far,

all variables are declared

after the opening brace

for a function

```
int main(void)
{
int count = 0;
float price = 4.59;
    // more code goes here
```

They're only accessible within the function itself

main()

doInventory()

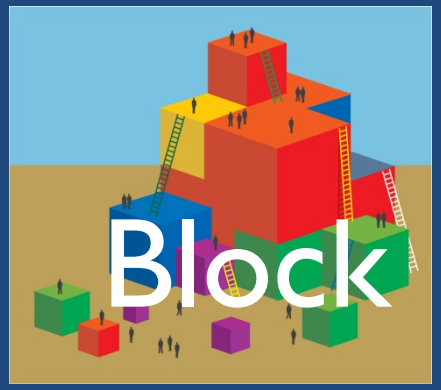
getNum()

displayReport()

General Scope Rule

"Identifiers are accessible only within the block in which they are declared. They are unknown outside the boundaries of that block." (page 213, ABoC)

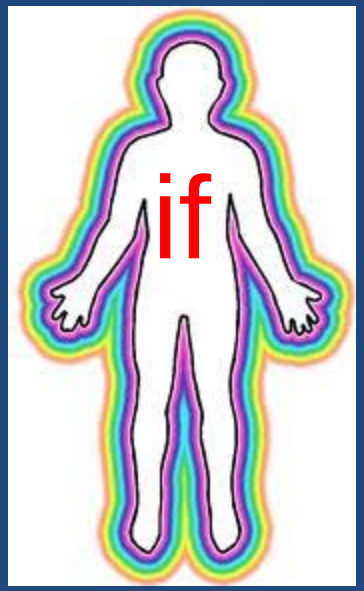
What's a Block?



{ Any code within curly braces }

a function body is
a block

||



an if statement's
body is a block,

||

etc.

Implication: You can have blocks within blocks.

Local Variables

Any variables declared within a function

or

any block within that function

are considered to be "local variables"

Also called

"auto variables"

created automatically when the block is entered

destroyed when the block is exited

This is a good thing.

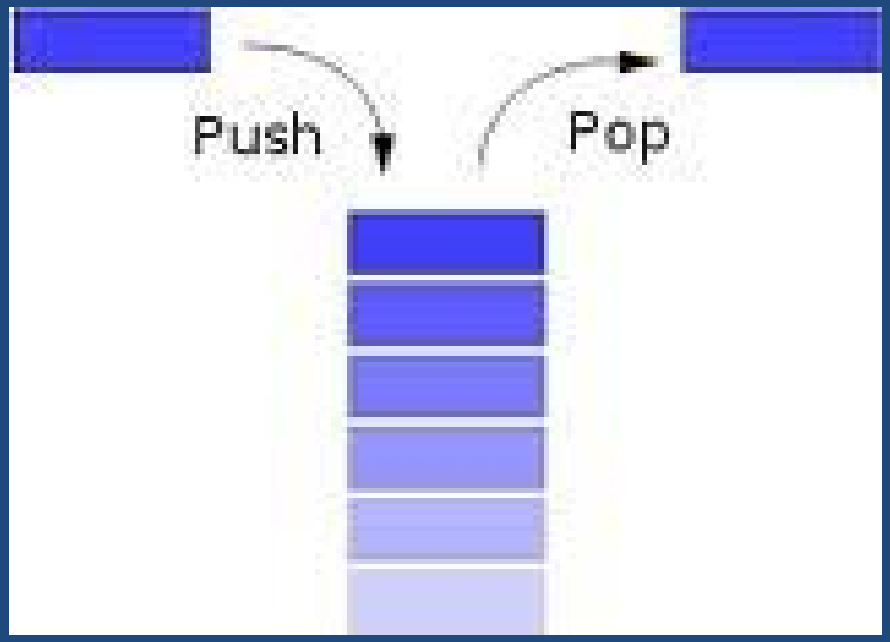
Why is it Good?

Reason #1: If you create variables only when you need them, you can save on memory.

Reason #2: If variables only exist close by to where they're used, they're easier to keep track of.

Where are they Created?

On the stack

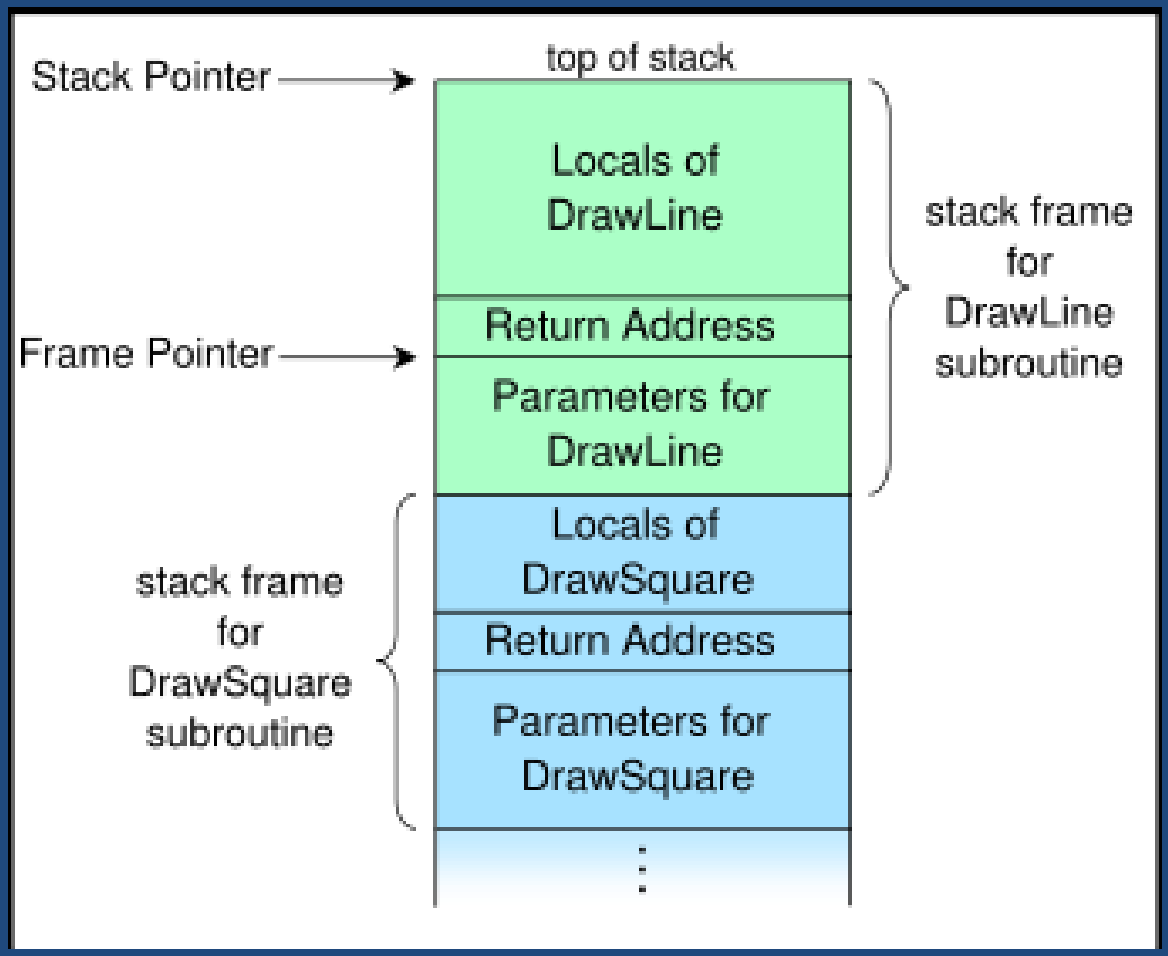


What is stored on the stack?

- auto variables
- parameters
- return addresses (for function calls)

How it's stored

Assume that DrawSquare() calls DrawLine():



From http://en.wikipedia.org/wiki/Call_stack

That's Why You Save Space

The stack space gets reused on every function call.

That's Why It's Important To Initialize Variables

```
int price;
```

float rate;

Uninitialized local variables

char letter;

get

garbage values!

Reason #2 Revisited

"If variables only exist close by to where they're used, they're easier to keep track of."

Keeping track of your data is a Very Good Thing.

Losing track of your data is a Very Bad Thing.

An Easy Way ...

Let's say that you could declare a variable once, somewhere in your program, and have it available everywhere.

everything everywhere™

You can, if you declare it outside of all functions.

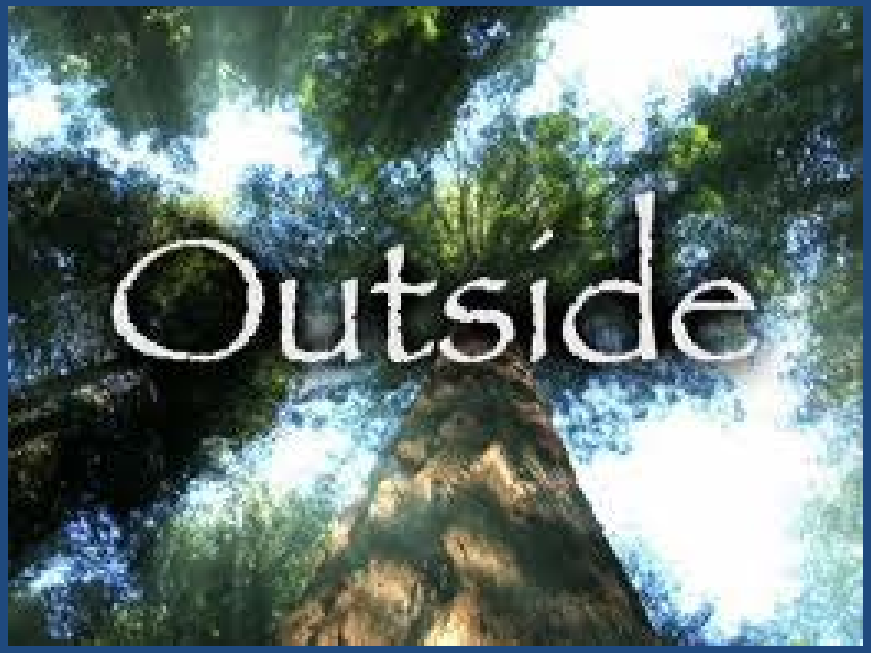
```
int thisIsAGlobalVariable = -13;

    int thisIsAFunction(void)
    {
          // the variable isn't here
    }
```

This creates a global variable (storage class extern).

Easy Global Variables

Global variables are:

declared outside of all functions



Global variables are:

created when the program starts

Global variables are:

accessible to any function

Soooo easy ...

e.g.

```
int howMany = 9;

void blah(void)
{
    printf("%d\n", howMany);
    goink();
    printf("%d\n", howMany);
}
```

howMany is a global variable.

What gets printed in this example?

Don't look ahead in the slides!

Any guesses?

The Answer!

The first printed line contains 9.

The second printed line contains 6314.

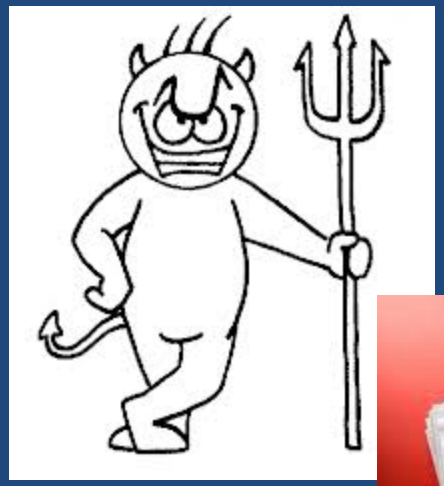
Eh?????

Proof:

- The goink() function is called.
  - goink() calls glort().
  - glort() calls zot().
  - zot() calls zoink().
- zoink() changes the value of howMany to 6314.

A BAD THING!!!

This behaviour is A Bad Thing.



The unpredictability of global variables make them very dangerous.


KILL THEM ALL!

Also, good design means learning to use functions with local variables and parameter passing.

Soooo ...

Course Use of Global Variables

The use of global variables in the course is banned.

- Exception: Assignment #1

The penalty can be up to ...

marks!

Yes, 50!

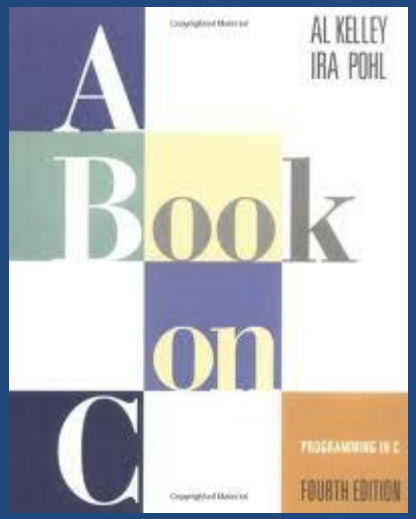
Caveat: Global variables are not always bad.

They have their place.

This course is not one of those places

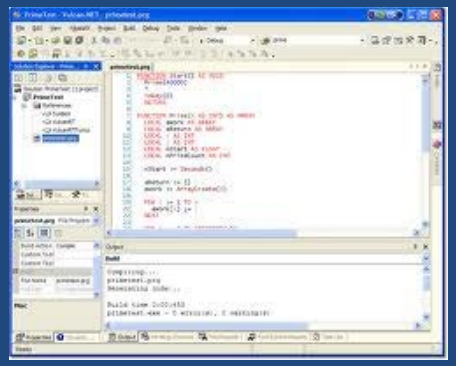
Other Storage Classes Besides Auto

Check out ABoC section 5.11 for other storage classes



Style

Why?

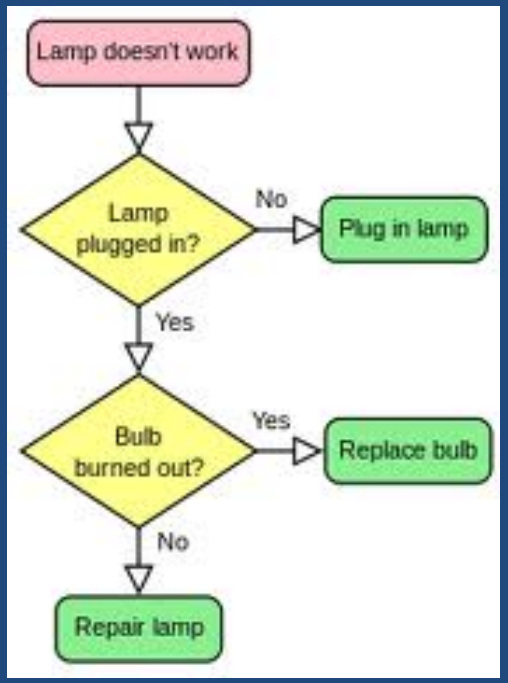


Style is not important to the compiler.

Style is important to the **programmer.**

From Code Complete ...

"Good visual layout shows the logical structure of a program."

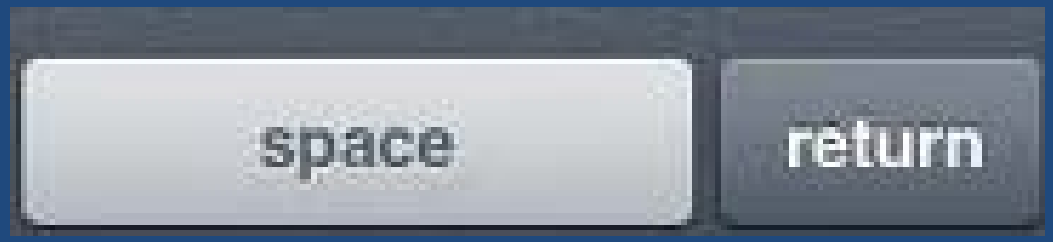


A good layout "should accurately and consistently represent the logical structure of the code, improve readability of the code, and withstand modifications."

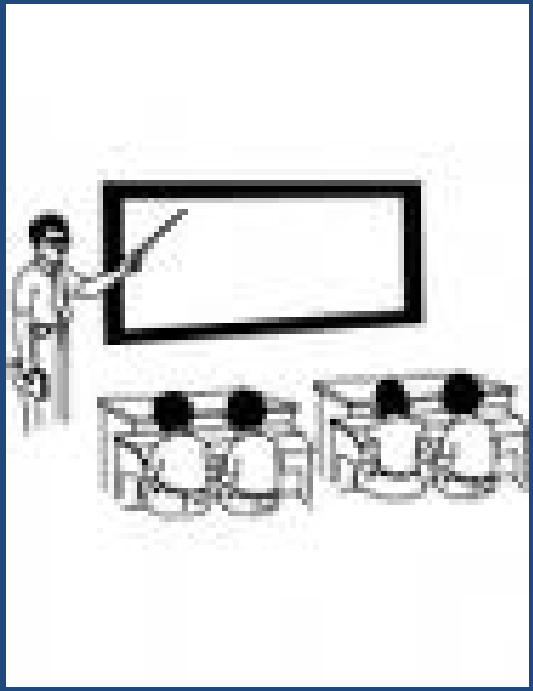
Consistency is vitally important!

Starting Out ...

Group related items together, using blank lines, spaces within lines, and indentation.

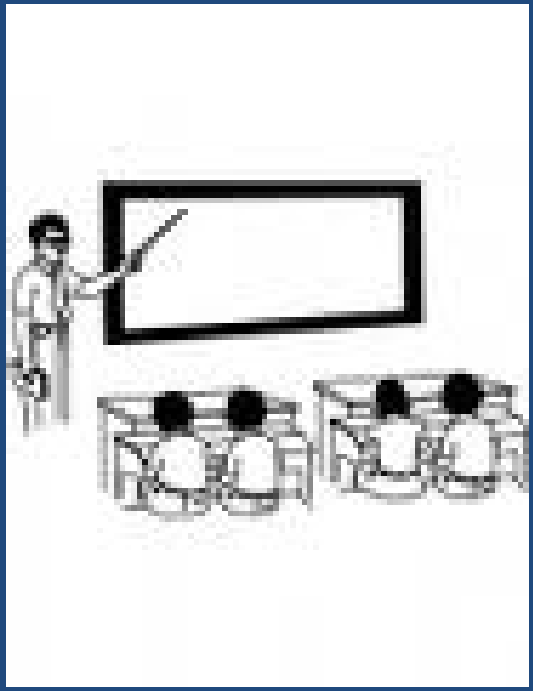


Let’s do an example



Indent the bodies of decisions (e.g. if bodies, else bodies, while bodies) to indicate a change of control flow.

Let’s do an example



- Usual indentation is one TAB or 3 to 5 spaces (TABs are easier).

- You MUST be consistent!

Caveat: On PowerPoints,
it's awkward to control
indentation,
so it might not look
consistent

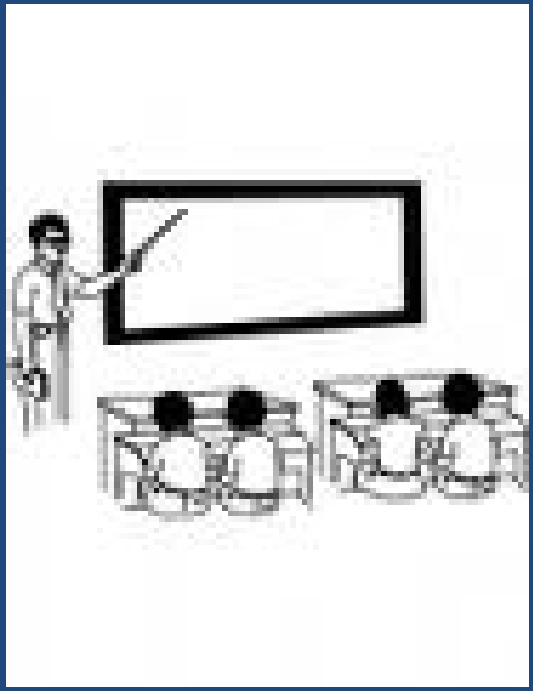
Indent the bodies of functions
so that they are

not level

with

the left margin

Let’s do an example



Vertical Whitespace

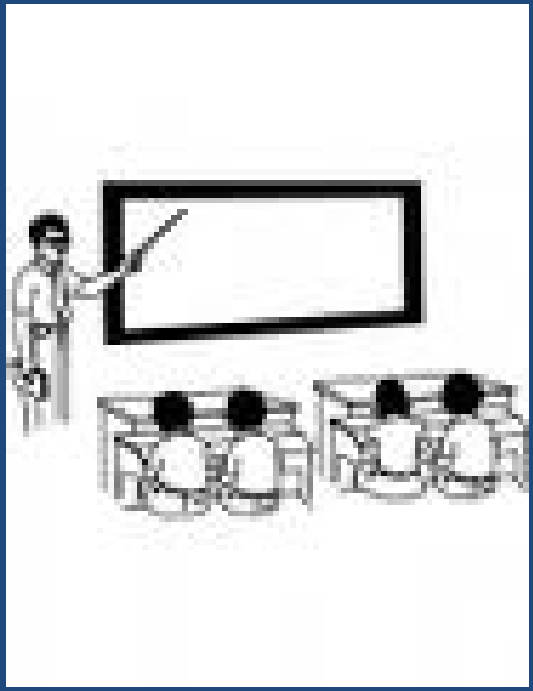
Put 3 to 5 consecutive blank lines

between major sections in your file

(e.g. #include, prototypes, functions).

Put 3 to 5 consecutive
blank lines
between
function definitions.

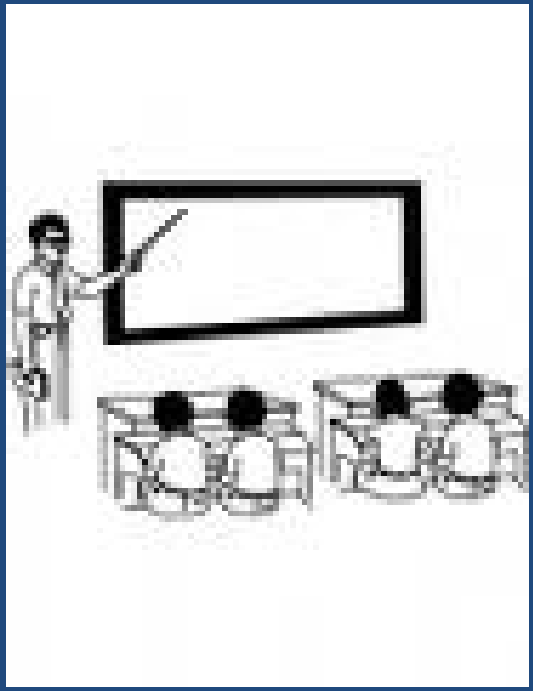
Let’s do an example



You would typically have one blank line:

- after variable declarations
- after the body of a control statement (not necessarily, though)
- between major sections of code

Let’s do an example

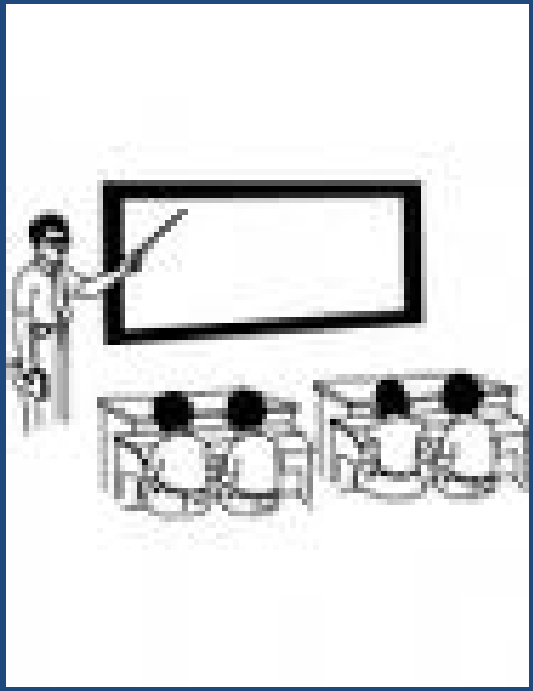


Horizontal Whitespace

It is common practice to put a single space:

- around arithmetic operators (e.g. =, +, -, etc.)
- after the commas in parameter and argument lists

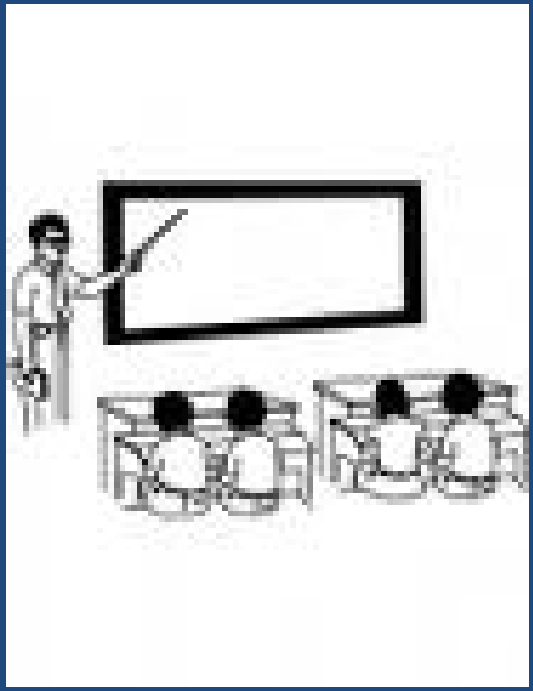
Let’s do an example



It is not unusual to put a single space:

- after or before round brackets

Let’s do an example



Three Bracing Styles

Style 1: Braces level with code block

e.g.

```
if( a == b )
    {
    printf("wow\n");
    }
```

Style 2: Braces level with control statement

e.g.

```
if( a == b )
{
    printf("wow\n");
}
```

Style 3: Braces attached to control statement (K&R Style)

e.g.

```
if( a == b ) {
    printf("wow\n");
}
```

Which one?

Religious Issue!

Choose one and use it throughout your program.

If you want to experiment,

do so on an assignment-by-assignment basis.

Ultimately, be flexible! You might have to change your style!

Inherent Bracing Inconsistency

If you're using
Style 2 (level w/control statement)
or
Style 3 (attached to control statement),
it is OK to indent variables
either:
with the braces
or
with the code block.

Style 2 Variable Declarations

```
int main(void)
{
int i = 0;      // OK
    // code goes here
}
```

```
int main(void)
{
    int i = 0;          // OK
    // code goes here
}
```

Style 3 Variable Declarations

```
int main(void) {
int i = 0;      // OK
    // code goes here
}
```

```
int main(void) {
    int i = 0;          // OK
    // code goes here
}
```

Array Initialization

We'll be learning about arrays soon.

We initialize arrays using braces.

In that case, it is fine to use whatever bracing you want to try and make the data most understandable.

Single Line Control Statement Bodies

As stated in week 2, you must put braces around single-line control statement bodies (e.g. if, else, while, etc.).

You should also **not** put the body on the same line as the control statement, even if it is trivial.

Why?

e.g. if(a == 0) a = 9;

Using the debugger, you can't get to the a = 9 line.

- Don't understand what I mean? ... wait until week 7.

Bracing else's

If using Style 1 or 2 (where the brace is **not** attached to the control statement),
it is generally considered acceptable to
either
put the braces above and below
or
on the same line as an else statement.

e.g.

```
if( i == 0 )
{
        i = 4;
}
else            // can also be        } else {
{
        i = 6;
}
```

Output Look and Feel in Assignments

Your output

should

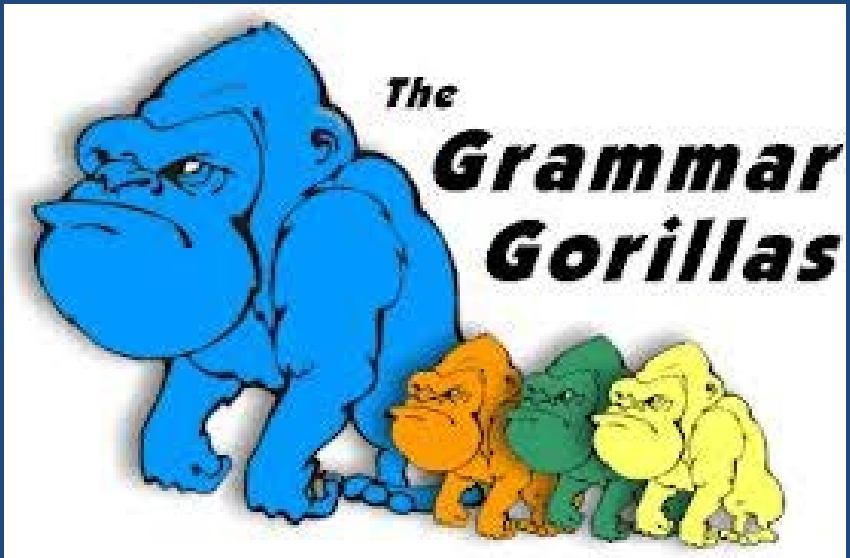
look

good

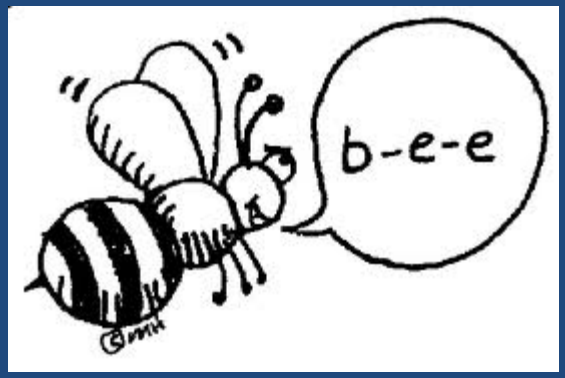
and work well

It should:

Be grammatically correct



Have no spelling mistakes

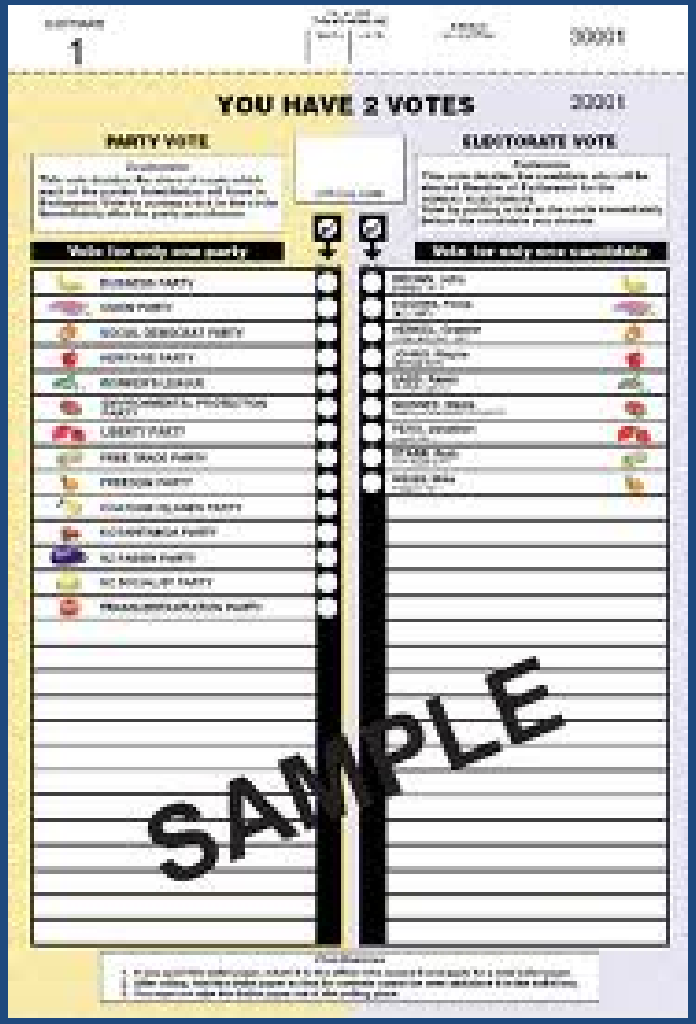


Be consistent

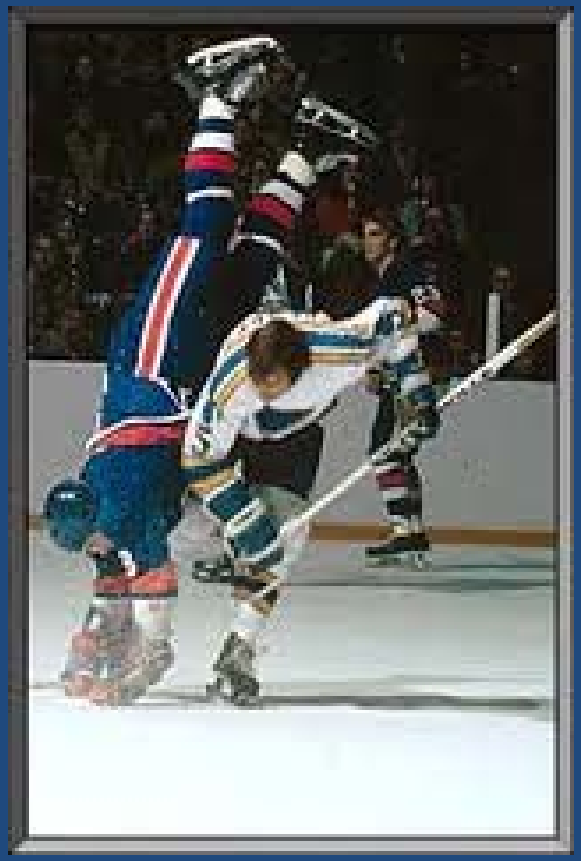
Look

good

Match any sample output (if stated)



Not
interfere
with the
user's usage
of your program



Finally!

All of these must be adhered to in Assignment #2

And ...

Many of these are also mentioned in the SET Coding Standards

Summary: Scope

- Local variable declarations happen at the start of C blocks
- Local variables are only accessible within their block

- Global variables can be declared but are not allowed in the course

Summary: Style

- Style is very important to programmers, not to the compiler
- Style is important in this course

- There is some flexibility but you need to follow established standards, including the SET Coding Standards