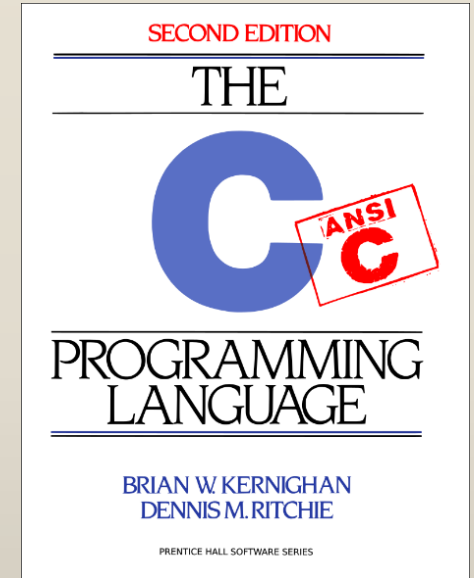
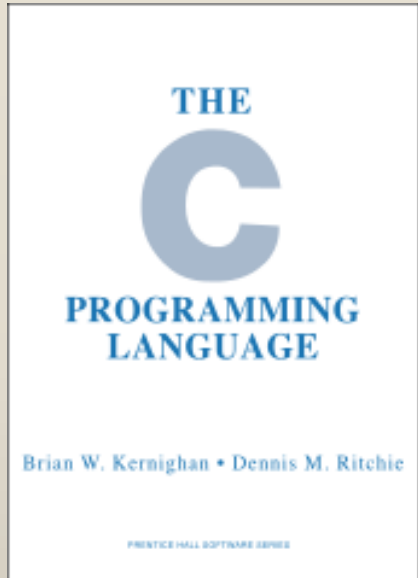




## Some basic C capabilities:

- Structures and `typedef`
- Scope and extent
- Pointers
- Strings
- `cpp` – the C Preprocessor
  - conditional compilation
  - macros
- `stdio`, `printf`, `fgets`, and others.



# Linux Programming in C

- Most programming on Linux is done in C (not C++, but plain C, **The Mother Tongue**).
- Most of the large programming assignments will be written in C (not C++, not Java, not Python).
  - You won't be able to use `cin` or `cout`. You will use `printf/fprintf` and `scanf/fscanf`.
  - You won't be able to use `new` or `delete`. You will use `malloc` and `free`.
- **You will also need to create a `Makefile` to build your C code for the labs.**



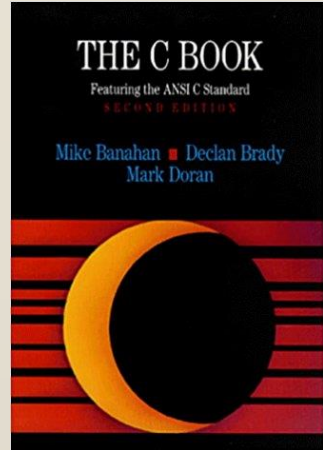
**VITAMIN**

# Linux Programming in C

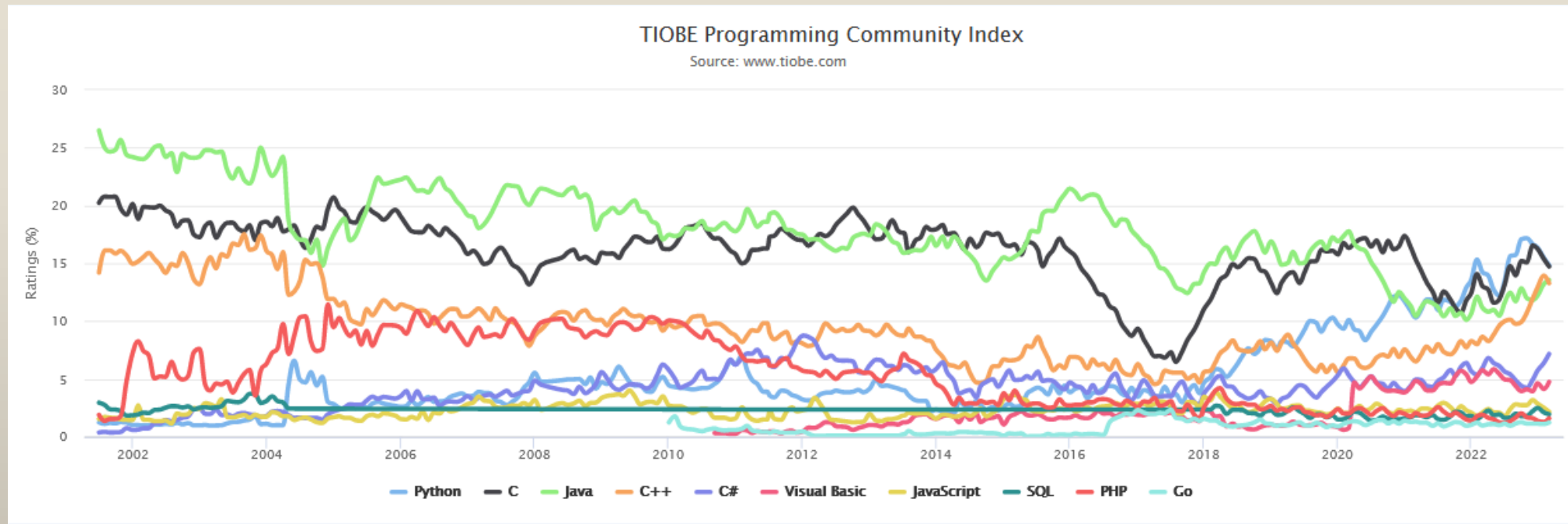
- C is a **systems programming** language, written to be able to manage system resources.
- By design, **C provides constructs that map efficiently to typical machine instructions**, and therefore it has found lasting use in applications that had formerly been coded in assembly language, **including operating systems**.
- It was **written by programmers for programmers**.
- C has been standardized by the American National Standards Institute (ANSI) since 1989.

# Resources for C

- I've placed links to a number of resources about using C and using C on Linux in the Canvas site under Class Resources/Resources for C .
- **I highly recommend you give them a look.**
- One specific resource is the completely free online book **The C Book** ([http://publications.gbdirect.co.uk/c\\_book/](http://publications.gbdirect.co.uk/c_book/)).
  - It is a little dated, but it contains most of everything you'll need for C programming in this class.
  - Of particular value is the page about the various formatting options for `printf()` and `scan()` calls [http://publications.gbdirect.co.uk/c\\_book/chapter9/formatted\\_io.html](http://publications.gbdirect.co.uk/c_book/chapter9/formatted_io.html)



This chart gives you an idea of how well the popularity of programming in C has held out over the years. C is the second most commonly used programming language.



<http://www.tiobe.com/tiobe-index/>

# Mathematical Expressions

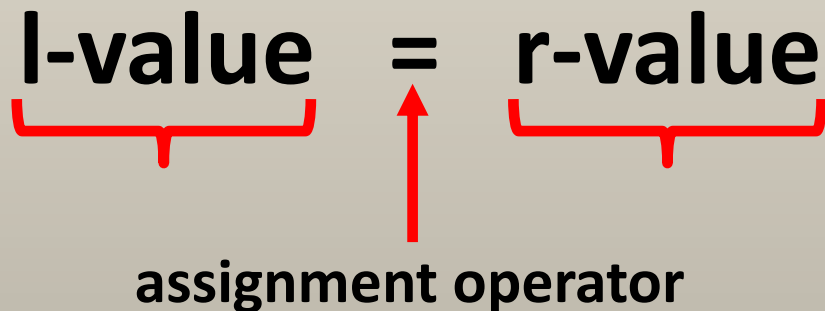
- ***l-value*** - refers to what can be placed on the **left of the assignment operator**

Constants and literals **cannot** be an l-value

***r-value*** - refers to what can be placed on the **right of the assignment operator**

Things like expressions and literals

**l-value**   **=**   **r-value**



assignment operator

```
int x = 0;  
x = ( 5 * 3 ) + 7;
```



# C Language Operators

Operator	Type	Description	Example
+	Binary	Addition	<code>a = b + 5;</code>
-	Binary	Subtraction	<code>a = b - 5;</code>
-	<b>Unary</b>	<b>Negation</b> (changes sign of value)	<b><code>a = -b;</code></b>
*	Binary	Multiplication	<code>a = b * 5;</code>
/	Binary	Division	<code>a = b / 5;</code>
%	Binary	<b>Modulus</b> (integer remainder of dividing left operand by right operand)	<b><code>a = b % 5;</code></b>



# The Modulus Operator

- Both operands of the **modulus** operator ( % ) **must** be integers.
- Modulus operator often causes confusion in beginning programmers.
- The integer remainder of dividing left operand by right operand.

Following results in value of 2

```
int var_a;  
var_a = 5 % 3;
```

The integer remainder of  $5 \div 3$



The integer remainder of the left operator divided by the right operator.

```
int var_a;  
var_a = 5 % 3;    // var_a equals 2  
var_a = 5 % 5;    // var_a equals 0  
var_a = 5 % 2;    // var_a equals 1  
var_a = 10 % 2;   // var_a equals 0  
var_a = 10 % 3;   // var_a equals 1  
var_a = 5 % 2;    // var_a equals 1  
var_a = 8 % 2;    // var_a equals 0
```

This is a common use of the modulus operator.  
Even or odd?

# Integer Division and Truncation

- When performing division, pay attention to the data types of the operands

- Examples:**

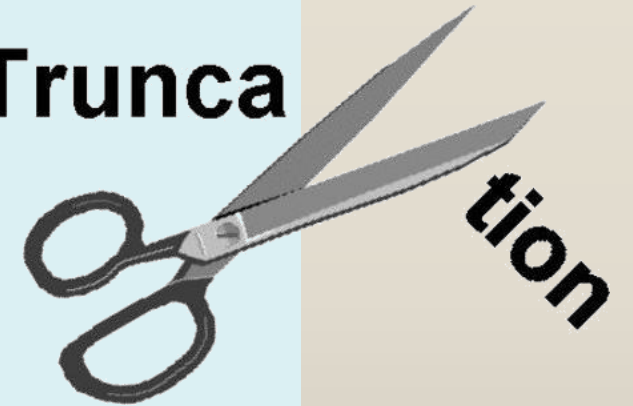
```
x = 5 / 9;
```

```
x = 5.0 / 9.0;
```

```
x = 5.0F / 9.0F;
```

Integer division  
could result in  
truncation

Trunca



- First example - performs integer division resulting in a zero being stored in x.**

PLEASE  
NOTE



- **Order of precedence** - established order that must be adhered to when evaluating expressions with multiple operations.
- The table lists common operators starting with those that will be evaluated first.



Performed first

Highest  
precedence

postfix ++, postfix --

prefix ++, prefix --, unary -

\*, /, %

+, -

=, \*=, /=, %=, +=, -=

Performed last

Lowest  
precedence

When in doubt...

**Parentheses are not endangered, use them.**

Relational Operator	Description
<b>==</b>	Equality ( <i>be sure to use two equal signs</i> )
<b>!=</b>	Inequality
<b>&lt;</b>	Less than
<b>&gt;</b>	Greater than
<b>&lt;=</b>	Less than or equal to
<b>&gt;=</b>	Greater than or equal to

- Single equal sign (=) is an assignment.
- Double equal sign (==) tests for equality.

This is a very common mistake, even for experienced developers.

**Truth table** - displays Boolean results produced when the operator is applied to specified operands.

Logical && and    truth table	C1	C2	C1 && C2	C1    C2
	true	true	true	true
	true	false	false	true
	false	true	false	true
	false	false	false	false

Logical NOT truth table	Condition	! ( Result )
	true	false
	false	true



# Short-circuit Evaluation

- **Short-circuit evaluation** - once the outcome of condition can be determined, evaluation ends.
  - If you can determine the left operand to an && operator is false, then short-circuit to false.
  - If you can determine that the left operand to a || operator is true, the short-circuit to true.

```
if ((b != 0) && ((a / b) > 100)) {  
    printf("Happy denominator\n");  
}  
else {  
    printf("Sad denominator\n");  
}
```

If b is equal to 0, the second half of the expression would cause an exception. Short-circuit evaluation prevents this.

# The Ternary Operator

- The ***Conditional operator*** - considered a **ternary** operator, meaning it has three operands.

- Syntax:

<condition> **?** <true expression> **:** <false expression>

Notice the question mark and the colon.

The ternary operator is basically an inline if-then-else statement.

# The Ternary Operator

This if/then/else statement can be replaced with the single line expression below.

```
if(Expression1) {  
    variable = Expression2;  
}  
else {  
    variable = Expression3;  
}
```

```
variable = Expression1 ? Expression2 : Expression3;
```

```
max_value = (n1 > n2) ? n1 : n2;
```

```
printf("The maximum value is %d\n", (n1 > n2) ? n1 : n2);
```

# Bitwise Operators

Operator	Description
&	Bitwise <b>AND</b> Operator copies a bit to the result if it exists in both operands. Note how this differs from the logical && operator.
	Bitwise <b>OR</b> Operator copies a bit if it exists in either operand. Note how this differs from the logical    operator.
^	Bitwise <b>XOR</b> Operator copies the bit if it is set in one operand but not both.
~	Bitwise <b>One's Complement</b> Operator is unary and has the effect of flipping bits.
<<	Bitwise <b>Left Shift</b> Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Bitwise <b>Right Shift</b> Operator. The left operands value is moved right by the number of bits specified by the right operand.

# Basic Data Types

	Data Type	Num Bytes (on os1)	Num Bits
Integral types	char	1	8
	short	2	16
	int	4	32
	long	8	64
	long long	8	64
Real types	float	4	32
	double	8	64
	long double	16	128
Pointer type	void *	8	64

The actual number of bytes (and therefore number of bits) for the types is not specified in the standard.

These are the sizes on the os1 server.

We will talk a LOT more about this one in another lecture.

The integral types can also be used with the unsigned qualifier:  
**unsigned short**  
**unsigned int**  
...

# Fixed-width Integer Types

Because the size of the integral data types was not part of the standard, an additional set of types was developed that are of known fixed size. These are defined in the include file `stdint.h`.

		Data Type	Num Bytes	Num Bits
		<code>int8_t</code>	1	8
		<code>int16_t</code>	2	16
		<code>int32_t</code>	4	32
		<code>int64_t</code>	8	64
unsigned	{	<code>uint8_t</code>	1	8
		<code>uint16_t</code>	2	16
		<code>uint32_t</code>	4	32
		<code>uint64_t</code>	8	64

# C 2-Dimensional Arrays

In C, a 2 dimensional array is actually implemented as an **array of arrays**.

```
int a[2][4];
```

A 2 dimensional array of integers, with 2 rows, each of which has 4 columns.

```
char c[12][20];
```

A 2 dimensional array of characters, with 12 rows, each of which has 20 columns.

```
float f[6][7];
```

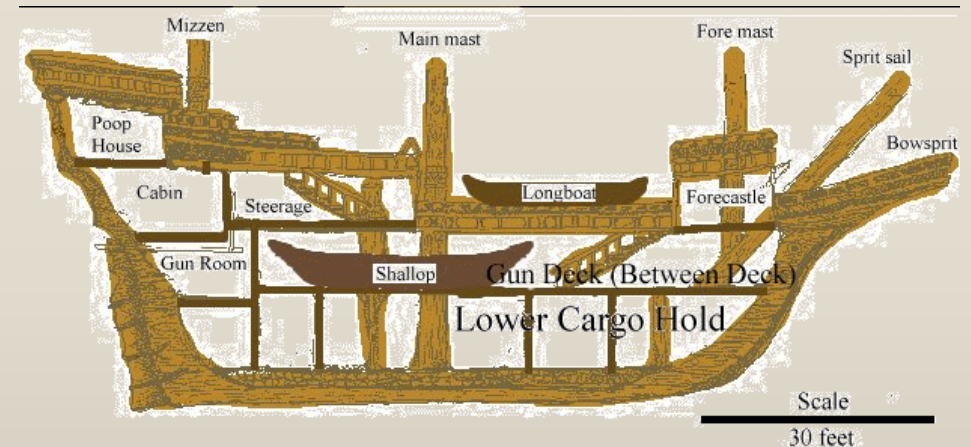
A 2 dimensional array of floating point values, with 6 rows, each of which has 7 columns.



# C Structures

This declares a structure called `account_s`. The `account_s` structure contains 4 members: `account_number`, `first_name`, `last_name`, and `balance`. In this example, the type of each member is a primitive type (`int`, `char`, or `float`).

```
struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
};
```



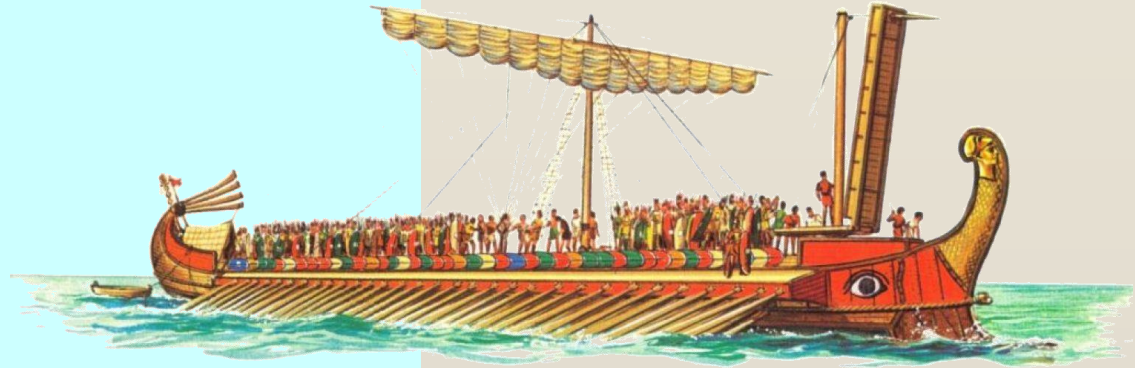
If you want to make use of the `account_s` structure in your code, you must declare a variable like this:

```
struct account_s personal_account;
```

The `struct` keyword must be used when declaring a variable from a structure.

# C Structures

```
struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
};
```



```
struct account_s personal_account;
```

If you want to access a member of the `account_s` structure, you need to prefix the member name with the structure variable name and use a dot before the member name.

```
personal_account.account_number = 309026;
```

The name of the variable we declared in of the `struct`.

The dot before the member name.

# C Structures and typedef

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```



If you want to skip the need to prefix the `struct` keyword when you declare a variable, you can `typedef` the structure to give it a new name. Now, you can declare a variable to be of type `account_t` as:

```
account_t personal_account;
```

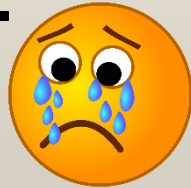
```
personal_account.account_number = 309026;
```

You access the variable members the same way.

As my own standard, I tend to use a `_s` as an ending on a structure and an ending of `_t` on a typedef.

# C Structures are **NOT** Classes

C structures are not classes.

- They **do not have** constructors.
- They **do not have** ante-constructors (aka destructors).
- They **do not have** methods.
- They **do not have** friends. 
- They **do have** data members, all of which are public.
- They **can be declared** to contain function pointers, but the function will not have a `this` pointer.

# C Structures Memory Layout

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```

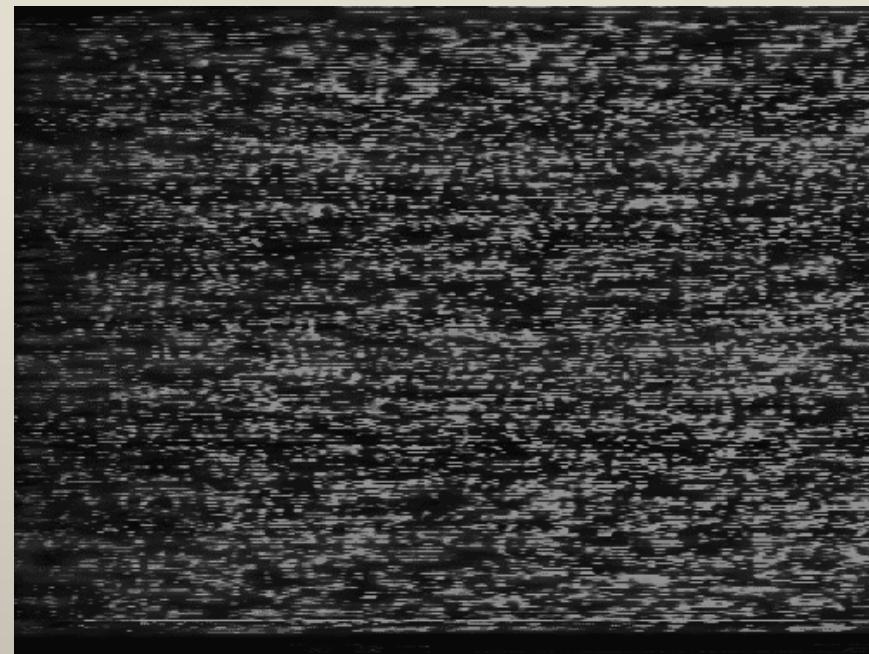
The memory for a C struct/typedef **will be laid out in the order that the members were specified in the struct definition.**

For the account\_s structure, the memory for the account\_number member will always be before the memory for the first\_name member.

And, the memory for the first\_name member will always be before the memory for the last\_name member.

# Some `static` Please

- In C, the reserved word ‘`static`’ has two, quite separate uses.
- The differences between the uses is often not well understood.
- In describing the `static` keyword, we first review the concept of scope.
- There are the concepts of **scope** and **extent**.
  - Scope refers to the *visibility* of variable or function.
  - Extent refers to the *lifetime* of a variable.





# Some static Please

- The variable **num\_occurrences** is defined outside the scope of any function (outside any braces).
  - Its scope (visibility) is global to the entire module (compilation unit or file).
  - If there are other modules that are part of the program, they cannot “see” the **num\_occurrences** variable. It has file/module scope.
- The lifetime (extent) of **num\_occurrences** is the life of the program.
  - It will be initialized once, at the start of the program.
  - As a global variable, the value in the **num\_occurrences** will persist for the lifetime of the program.
  - The **num\_occurrences** will be stored in the data segment.

```
static int num_occurrences = 0;
```

```
int main()  
{  
    int value = 0;  
  
    return 0;  
}
```

- Contrast that to the variable `value`, which is local only to `main`.
- The `value` variable will be stored on the stack and will be deallocated from the stack when the function returns.



# Some `static` Please

- The variable **`num_occurrences`** is defined locally within the function.
  - Its scope (visibility) is strictly within the function.
  - Its lifetime (extent) is the life of the program.
  - It does not get recreated, on the stack, each time the function is called.
- It will be initialized once, at the start of the program.
  - The value in the **`num_occurrences`** will persist beyond the lifetime of an individual invocation of the function.
  - The **`num_occurrences`** will be stored in the data segment.

```
int count_occurrences(int num_to_add)
{
    static int num_occurrences = 0;
    int value = 0;

    num_occurrences += num_to_add;

    return num_occurrences;
}
```

- Contrast that to the variable `value`, which will be created and initialized each time the function is called.
- The `value` variable will be stored on the stack and will be deallocated from the stack when the function returns.

# Some static Please

- The function `blaa1` has **external linkage**. Its name is visible to functions both within and outside the module (compilation unit or file).
- The function `blaa2` has **internal linkage**. Its name is only visible from within the module where it is defined.
- The scope of the `blaa1` function is global to the program.
- The scope of the `blaa2` function is local to the module.
  - In C++ terms, we can say that the `blaa2` function is private to the file.

The extent (lifetime) of both functions is the lifetime of the program.

```
int blaa1(int num_to_add)
{
    static int value = 0;

    value += num_to_add;

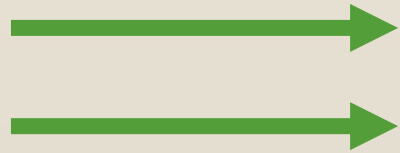
    return value;
}

static int blaa2(int num_to_add)
{
    static int value = 0;

    value += num_to_add;

    return value;
}
```

# C Pointers



# C Pointers



```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```

Let's start off with the previous simple structure typedef.

Now we'll declare 2 variables:

```
account_t personal_account;  
account_t *personal_account_ptr = &personal_account;
```

Declare a **pointer** to something of type `account_t`.

This is how you get the address of a variable. The address of the variable is then a pointer back to that variable. **You will do this in this class.**

Initialize it to be the address of the other variable.

# C Pointers

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```



Now assign a value to the `account_number` member, we can use the pointer syntax to access the member.

```
personal_account_ptr->account_number = 309026;
```

The pointer variable we declared in the previous slide.

You could use the other, **more cumbersome**, syntax of

```
*personal_account_ptr.account = 309026;
```

But, why? Which has a higher precedence, the `.` or the `*`



# Strings in C

Things you may **think** you want to do:

```
char *str1;  
char str2[25] = "hello ";  
char str3[25] = "world";
```

```
str1 = "this is silly";  
str3 = str2;  
str2 = str2 + str3;  
  
if ( str2 == str3 ) {  
    ...  
}
```

What will the result of  
this comparison be?

Nono Bad dog!

Why are these wrong?

HI!! My name is

NONO BAD DOG!

What's yours?

In C++ terms, these are cStrings,  
**not the C++ string class.**

# Strings in C

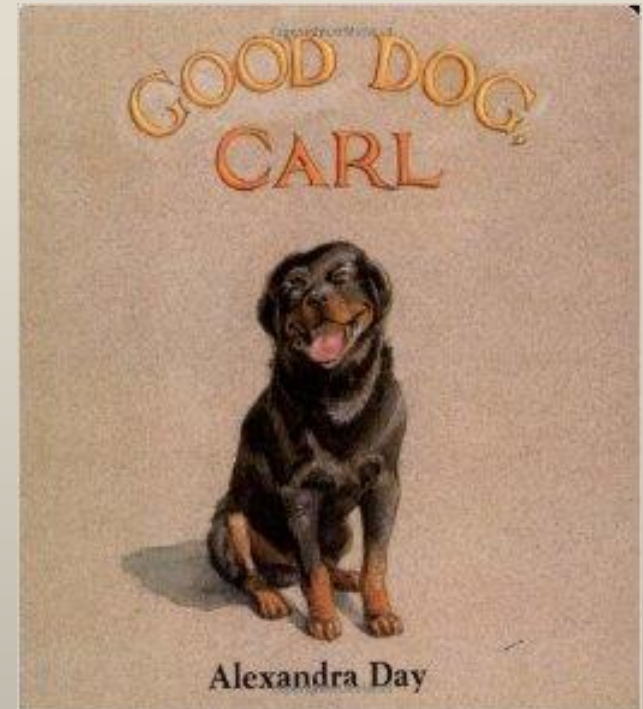
Things you **will** to do:

```
char *str1;  
char str2[25] = "hello ";  
char str3[] = "world";
```

We'll cover more about  
`strdup()` in another lecture.

```
str1 = strdup("this is silly");  
strcpy(str3, str2);  
strcat(str2, str3);  
  
if (strcmp(str2, str3)) {  
    ...  
}
```

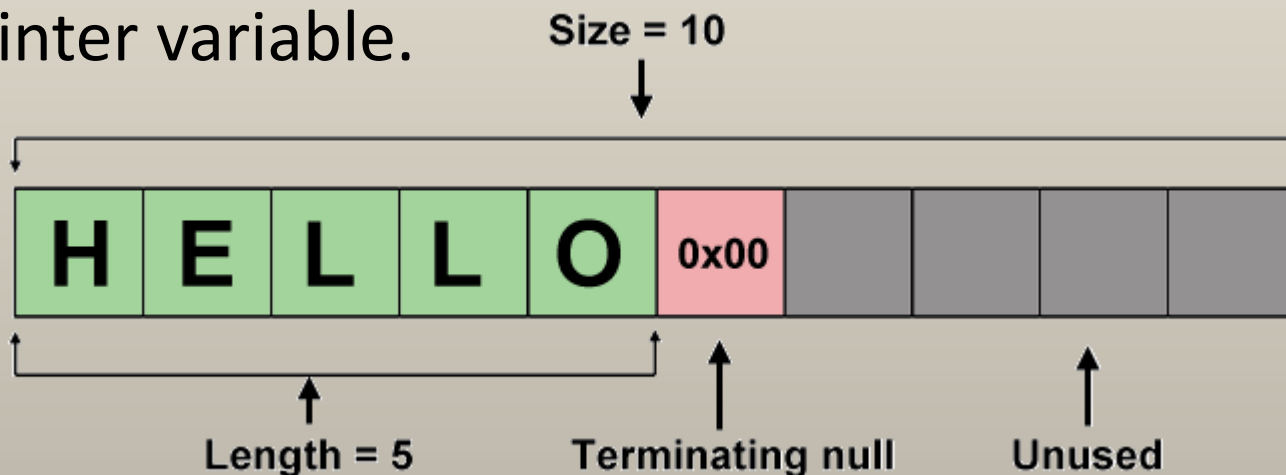
What are the return values  
for `strcmp()`?





# Strings in C

- In C, a string is a null-terminated array of bytes of type `char`, including the terminating null byte.
- String-valued variables are often declared to be **pointers** of type `char *`.
- **Such pointer variables do not include space for the text of a string;** that has to be stored somewhere else.
- It is up to you to store the address of the chosen memory space into the pointer variable.



# Strings in C

- When creating a **string literal in C**, it must be within double quote marks.

"this is a string literal"

"A"

""

These are valid C string literals.

- If you want to create a **single character literal**, you can use single quotes, but it is not a string. It is a single `char` literal.

'A'

// This is a character literal, not a string literal.

Single quotes means single character.  
Double quotes can be multiple characters.

# The C String Functions

A partial list of the C String Functions:

- `strcat()`
- `strncat()`
- `strcmp()`
- `strncmp()`
- `strcpy()`
- `strncpy()`
- `strlen()`

Notice that several of these have the variant call with the n.



[https://en.wikibooks.org/wiki/C\\_Programming/Strings](https://en.wikibooks.org/wiki/C_Programming/Strings)

Append from `src` into `dest`.

```
#include <string.h>
```

```
char *strcat(char *dest, const char *src);
```

```
char *strncat(char *dest, const char *src, size_t n);
```



- The `strcat()` function **appends** the `src` string to the `dest` string, overwriting the terminating null byte (`'\0'`) at the end of `dest`, and then adds a terminating null byte.
- If `dest` is not large enough, program behavior is unpredictable.
- The `strncat()` function is similar, except that it will use at most `n` bytes from `src`.

```
#include <string.h>
```

These do not return  
true or false.

```
int strcmp(const char *s1, const char *s2);
```

```
int strncmp(const char *s1, const char *s2, size_t n);
```

- The `strcmp()` function compares the two strings `s1` and `s2`.
- It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.
- The `strncmp()` function is similar, except it compares only the first (at most) `n` bytes of `s1` and `s2`.

<code>int strcmp(s1, s2) ;</code>	Return value	
<code>s1 &lt; s2</code>	<code>&lt; 0</code>	Notice that this is not equal to -1, rather less than 0.
<code>s1 == s2</code>	<code>0</code>	
<code>s1 &gt; s2</code>	<code>&gt; 0</code>	Notice that this is not equal to 1, rather greater than 0.

The `strcmp` function compares the string contents, not the pointer values.



Copy from `src` into `dest`.  
Think of the comma as an equal sign.

```
#include <string.h>
```

```
char *strcpy(char *dest, const char *src);
```

```
char *strncpy(char *dest, const char *src, size_t n);
```

- The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`.
- The destination string `dest` must be large enough to receive the copy.
- The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied.
  - If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null-terminated.





```
#include <string.h>
```

```
size_t strlen(const char *s);
```



The `strlen()` function calculates the length of the string pointed to by `s`, excluding the terminating null byte (`'\0'`).

The `strlen()` function returns the number of characters in the string pointed to by `s`.

If you call `strlen()` on something that is not NULL terminated, you will get unpredictable results.

Here is a test question you will see:

Does `strlen()` include the NULL character in the length of a string?



# Strings in C

A closing note about chars and strings in C.

Just because something is an array of `char` (or a pointer to `char`), it does not mean that it is a string.

If the `char` array is not `NULL` terminated, it is not a string, it is just an array of characters.

## Not all character data are strings.

Don't get tied up in knots over this.



# Strings in C

Here is a test question you will see:

In C, what is the difference between an array of `char` and a string?

