CSE2421 = Alice in Wonderland

A look into the rabbit hole...
Introduction

- Website
  - Personal: [http://www.cse.ohio-state.edu/~reeves](http://www.cse.ohio-state.edu/~reeves)

- Syllabus
  - Course Description
  - Pre-requisites
  - Objectives
  - Textbook – Safari online an option for C programming
  - Grading Policy
  - Academic Misconduct

- Day2 lab day in BE0310 (except summer term)
  - Lab locations

- Discussion Group - Piazza

- Grade Posting – by code given
Why C?

- Age has its advantages
  - C has been around for ~40 years
- C is a great language for expressing common ideas in programming in a way that most people are comfortable with (procedural language)
- Portable, versatile, simple, straight-forward
- Reasonably close to the machine
  - Low-level access to memory
  - Provide language constructs that map efficiently to machine instructions
  - Requires minimal run-time support

*** C has the best combination of speed, low memory use, low-level access to the hardware, and popularity ***

OK, really... why C?

- Is there a size problem?
  - Size is part of the issue, but so is speed.
  - C is lightweight and fast.

- I love garbage
  - No automatic garbage collection
  - Fun memory leaks to debug

- Wonderfully, yet irritatingly, obedient
  - You type something incorrectly, and it has a way of compiling fine and just doing something you don't expect at run-time.

- Power...
  - To optimize
  - Write drivers
  - Get a job in micro processing technology
  - Write my own OS
Going from Java to C is like going from an automatic transmission to a stick shift
- Lower level: much more is left for you to do
- Unsafe: you can set your computer on fire
- C standard library: is much smaller
- Similar syntax: can both help and confuse
- Not object oriented: paradigm shift
Happiness is... programming in C

- C is procedural, not object-oriented
- C is fully compiled (to machine code), not to bytecode
- C allows direct manipulation of memory via pointers
- C does not have garbage collection
- Many of the basic language constructs in C act in similar ways to the way they work in Java
- C has many important, yet subtle, details
C vs Java/C++

Programming language rankings
Speed - Portability - Object Orientation

- Pointers to memory
- Platform dependent types
- Programmer allocated memory
- Declare variables at start of block

- References to objects
- Types have well defined sizes
- Automatic garbage collection
- Declare variable anywhere
C does not...

- C is a procedural language, and *does not support objects*. That is, it does not support entities which contain data and model behavior. We can botch together something of the sort in C, but it is still far from what we would ever consider a class.

- C *does not support Encapsulation*. While you may set up a group of data types to only be accessible through a structure collection, it still can be accessed from anywhere, by anything, as long as the collection exists within a scope seen by what is trying to access it.
C History and background

- **What is C?**
  - C is a programming language originally created for developing the Unix operating system. It is a low-level and powerful language, but it lacks many modern and useful constructs.
  - C is a simple programming language with few keywords and a relatively simple to understand syntax.
  - C is also useless (whaaaaaaannnnnnnttt???). C itself has no input/output commands, doesn't have support for strings as a fundamental (atomic) data type. No useful math functions built in.
  - Because C is useless by itself, it requires the use of libraries. This increases the complexity of C. The issue of standard libraries is resolved through the use of ANSI libraries and other methods.

- **Three traditional aspects of the C language:**
  - Characters are promoted to integers before being used for any type of arithmetic.
  - The default character type, either signed or unsigned, is not specified by the Standard so that the implementer can choose whichever is most efficient for a particular machine.
  - There is no range checking on array subscripts.
BRIAN KERNIGHAN QUOTES

- Controlling complexity is the essence of computer programming.
  
  "Software Tools" (1976), p. 319 (with P. J. Plauger)

- The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.
  
  "Unix for Beginners" (1979)

- Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?
  
  "The Elements of Programming Style", 2nd edition, chapter 2

- Do what you think is interesting, do something that you think is fun and worthwhile, because otherwise you won't do it well anyway.
  
  An Interview with Brian Kernighan from the PC Report Romania[1]

- Advice to students: Leap in and try things. If you succeed, you can have enormous influence. If you fail, you have still learned something, and your next attempt is sure to be better for it.

- Advice to graduates: Do something you really enjoy doing. If it isn’t fun to get up in the morning and do your job or your school program, you’re in the wrong field.

  "Leap In and Try Things: Interview with Brian Kernighan"[2] from Harmony at Work blog[3]
Your first C program

Reminder → There are a lot of different ways to solve the same problem.

TO-DO: Experiment with leaving out parts of the program, to see what error messages you get.
C compilation model... hello.c to hello

Type in program using an editor of your choice (file.c); plain text

.c + .h = .i which is the “ultimate source code”? i.e. # includes expanded and #defines replaced

.i → .s which is assembler source code

.s → .o which is an object file; fragments of machine code with unresolved symbols i.e. some addresses not yet known (vars/subrs).

.o + library links → a.out (default name); resolves symbols, generates an executable.

%gcc -o hello hello.c

%hello
Functions, types and macros of the standard library are declared in standard headers:

- `<assert.h>`  
- `<float.h>`  
- `<math.h>`  
- `<stdarg.h>`  
- `<stdbool.h>`  
- `<stdlib.h>`  
- `<ctype.h>`  
- `<limits.h>`  
- `<setjmp.h>`  
- `<errno.h>`  
- `<locale.h>`  
- `<signal.h>`  
- `<stdio.h>`  
- `<time.h>`

A header can be accessed by

- `#include <header>`
- Notice, these do not end with a semi-colon

Headers can be included in any order and any number of times

Must be included outside of any external declaration or definition; and before any use of anything it declares

Need not be a source file
A header file is a file containing C declarations and macro definitions (see Macros) to be shared between several source files. You request the use of a header file in your program by including it, with the C preprocessing directive `#include'.

Header files serve two purposes.
- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

In C, the usual convention is to give header files names that end with .h. It is most portable to use only letters, digits, dashes, and underscores in header file names, and at most one dot.
Every full C program begins inside a function called "main". A function is simply a collection of commands that does "something". The main function is always called when the program first executes. From main, we can call other functions, whether they be written by us or by others or use built-in language features.

A C program must have at least one function; and that function is called `main`.

Java programmers may recognize the main() method but note that it is not embedded within a class. C does not have classes. All methods (simply known as `functions`) are written at file scope.

The main() method in Java has the prototype ‘main(String[] args)’ which provides the program with an array of strings containing the command-line parameters. In C, an array does not know its own length so an extra parameter (argc) is present to indicate the number of entries in the argv array.
Your first C program (cont)

- What is going on?
  - `#include <stdio.h>` - Tells the compiler to include this header file for compilation. To access the standard functions that comes with your compiler, you need to include a header with the `#include` directive.
    - What is a header file? They contain prototypes and other compiler/pre-processor directives. Prototypes are basic abstract function definitions. More on these later...
    - Some common header files are `stdio.h`, `stdlib.h`, `unistd.h`, `math.h`.
  - `main()` - This is a function, in particular the main block.
  - `{ }` - These curly braces are equivalent to stating "block begin" and "block end". The code in between is called a “block”
  - `printf()` - Ah... the actual print statement. Thankfully we have the header file `stdio.h`! But what does it do? How is it defined?
  - `return 0` - What's this? Every function returns a value...
    - you should always explicitly declare the return type on the function. **If you don’t, it defaults to a type integer anyway.**
The return 0 statement. Seems like we are trying to give something back, and it is an integer. Maybe if we modified our main function definition: int main() Ok, now we are saying that our main function will be *returning* an integer! So remember, you should always explicitly declare the return type on the function! **If you don’t, it defaults to a type integer anyway.**

Something is still a little fishy... I thought that 0 implied false (which it does)... so isn't it returning that an int signifying a bad result? Thankfully there is a simple solution to this. Let's add `#include <stdlib.h>` to our includes. Let's change our return statement to return `EXIT_SUCCESS;`. Now it makes sense!

Let's take a look at `printf`. Hmm... I wonder what the prototype for `printf` is. (btw, what’s a prototype?) Utilizing the man pages we see that `printf` is: `int printf(const char *format, ...);` `printf` returns an int. The man pages say that `printf` returns the number of characters printed. Now you wonder, who cares? Why should you care about this? It is good programming practice to **ALWAYS** check for return values. It will not only make your program more readable, but in the end it will make your programs less error prone. But in this particular case, we don't really need it. So we cast the function's return to (void). `fprintf`, `fflush`, and `exit` are the only functions where you should do this. More on this later when we get to I/O. For now, let's just void the return value.

What about **documentation**? We should probably doc some of our code so that other people can understand what we are doing. Comments in the C89 standard are noted by: `/* */`. The comment begins with `/*` and ends with `*/`.

- **Comments cannot be nested!**
- `//` is a single line comment i.e. from the location of `//` to the end of the line is considered a comment
Much better! The **KEY POINT** of this whole introduction is to show you the fundamental difference between **correctness** and **understandability**. All of the sample codes produce the exact same output in "Hello, world!" However, only the latter example shows better readability in the code leading to code that is understandable. All codes will have bugs. If you sacrifice code readability with reduced (or no) comments and cryptic lines, the burden is shifted and magnified when your code needs to be maintained.
NOTE: There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, form feeds and comments (collectively, “white space”) are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.
## Basic Data Types

- **Integer Types**
  - Char – smallest addressable unit; each byte has its own address
  - Short – not used so much
  - Int – default type for an integer constant value
  - Long – do you really need it?

- **Floating point Types** – are “inexact”
  - Float – single precision (about 6 digits of precision)
  - Double – double precision (about 15 digits of precision)
    - Constant default unless suffixed with ‘f’

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2 bytes</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4 bytes</td>
<td>0 to 4294967295</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4 bytes</td>
<td>0 to 4294967295</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>1.175494e-38 to 3.402823e+38</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>2.225074e-308 to 1.797693e+38</td>
</tr>
</tbody>
</table>

Note that variables of type char are guaranteed to always be one byte.

There is no maximum size for a type, but the following relationships must hold:

- `sizeof(short) <= sizeof(int) <= sizeof(long)`
- `sizeof(float) <= sizeof(double) <= sizeof(long double)`
Derived types (fyi)

Beside the basic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- **arrays** of objects of a given type;
- **functions** returning objects of a given type;
- **pointers** to objects of a given type;
- **structures** containing a sequence of objects of various types;
- **unions** capable of containing any of one of several objects of various types.

In general these methods of constructing objects can be applied recursively:

- An array of pointers
- An array of characters (i.e. a string)
- Structures that contain pointers
Constants

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>'A', 'B'</td>
</tr>
<tr>
<td>int</td>
<td>123, -1, 2147483647, 040 (octal), 0xab (hexadecimal)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>123u, 2107433648, 040U (octal), 0X02 (hexadecimal)</td>
</tr>
<tr>
<td>long</td>
<td>123L, 0x1FFFF1 (hexadecimal)</td>
</tr>
<tr>
<td>unsigned long</td>
<td>123ul, 0777UL (octal)</td>
</tr>
<tr>
<td>float</td>
<td>1.23F, 3.14e+0f</td>
</tr>
<tr>
<td>double</td>
<td>1.23, 2.718281828</td>
</tr>
<tr>
<td>long double</td>
<td>1.23L, 9.99E-9L</td>
</tr>
</tbody>
</table>

Special characters

- Not convenient to type on a keyboard
- Use single quotes i.e. `\n`
- Looks like two characters but is really only one

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\a</code></td>
<td>alert (bell) character</td>
</tr>
<tr>
<td><code>\b</code></td>
<td>backspace</td>
</tr>
<tr>
<td><code>\f</code></td>
<td>formfeed</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>newline</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>carriage return</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>horizontal tab</td>
</tr>
<tr>
<td><code>\v</code></td>
<td>vertical tab</td>
</tr>
<tr>
<td><code>\\</code></td>
<td>backslash</td>
</tr>
<tr>
<td><code>\?</code></td>
<td>question mark</td>
</tr>
<tr>
<td><code>\\</code></td>
<td>single quote</td>
</tr>
<tr>
<td><code>\&quot;</code></td>
<td>double quote</td>
</tr>
<tr>
<td><code>\000</code></td>
<td>octal number</td>
</tr>
<tr>
<td><code>\xhh</code></td>
<td>hexadecimal number</td>
</tr>
</tbody>
</table>
Symbolic constants

- A name that substitutes for a value that cannot be changed
- Can be used to define a:
  - Constant
  - Statement
  - Mathematical expression
- Uses a preprocessor directive
  - `#define <name> <value>`
    - No semi-colon
  - Coding style is to use all capital letters for the name
- Can be used any place you would use the actual value
- All occurrences are replaced when the program is compiled
- Examples (see link below):
  - The use of EXIT_SUCCESS in hello.c code (see stdlib.h)
  - `#define PI 3.141593`
  - `#define TRUE 1`
  - `#define floatingpointnum float`

http://www.cprogrammingexpert.com/C/Tutorial/fundamentals/symbolic_constant_c_programming_language.aspx
### Variable Declarations

**Purpose:** define a variable before it is used.

**Format:** `type identifier [, identifier] ;`

**Initial value:** can be assigned

- `int i, j, k;`
- `char a, b, c = 'D';`
- `int i = 123;`
- `float f = 3.1415926535;`
- `double f = 3.1415926535;`
- strings later... array of characters

**Type conversion:** *aka*, type casting

- Coercion, be very cautious.
- `(type) identifier;`
  - `int i = 65; /* what if 258 */`
  - `char a; /* range -128 to 127 */`
  - `a = (char) i; /* What is the value of a? */`

---

**INSTANCE VARIABLE**

In object-oriented programming with classes, an *instance variable* is a variable defined in a class (i.e. a member variable), for which each object of the class has a separate copy. They live in memory for the life of the class.
Variable vs Identifier

- An identifier, also called a token or symbol, is a lexical token that “names” an entity
  - An entity can be: variables, types, labels, functions, packages, etc.
  - Naming entities makes it possible to refer to them

- A variable
  - Allows access and information about what is in memory i.e. a storage location
  - A symbolic name (an identifier) that is associated with a value and whose associated value may be changed
  - The usual way to reference a stored value
Identifier Naming Style

- **Rules for identifiers**
  - a-z, A-Z, 0-9, and _
  - Case sensitive
  - The first character must be a letter or _
  - Keywords are reserved words, and may not be used as identifiers

- **Identifier Naming Style**
  - Separate words with ‘_’ or capitalize the first character
  - Use all UPPERCASE for symbolic constant, macro definitions, etc
  - Be consistent
  - Be meaningful

- **Sample Identifiers**
  - i0, j1, abc, stu_score, __st__, data_t, MAXOF, MINOF...
Keywords

<table>
<thead>
<tr>
<th>auto</th>
<th>double</th>
<th>int</th>
<th>struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>else</td>
<td>long</td>
<td>switch</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>register</td>
<td>typedef</td>
</tr>
<tr>
<td>char</td>
<td>extern</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>continue</td>
<td>for</td>
<td>signed</td>
<td>void</td>
</tr>
<tr>
<td>default</td>
<td>goto</td>
<td>sizeof</td>
<td>volatile</td>
</tr>
<tr>
<td>do</td>
<td>if</td>
<td>static</td>
<td>while</td>
</tr>
</tbody>
</table>

Purpose: reserves a word or identifier to have a particular meaning

- The meaning of keywords — and, indeed, the meaning of the notion of *keyword* — differs widely from language to language.
- You shouldn't use them for any other purpose in a C program. They are allowed, of course, within double quotation marks.
Determine if the following are valid or invalid variable names. Explain if they are not.

1. goto
2. GoTo
3. main
4. ___
5. ca$h
6. abcdefghijklmnopqrstuvwxyz123456789
7. abcdefghijklmnopqrstuvwxyz123456
8. printf
9. avg value
10. a*b
11. 123
12. bukeye
Mathematical Symbols
+ - * / %
addition, subtraction, multiplication, division, modulus
Works for both int and float
+ - * /
/ operator performs integer division if both operands are integer
i.e. truncates; otherwise, float
% operator divides two integer operands with an
integer result of the remainder
Precedence – left to right
() always first
* / %
+ -
In C, assignments are expressions, not statements. Embedded assignments - legal anywhere an expression is legal. This allows multiple assignment `a = b = c = 1;` Assignment operators
- Same precedence; right to left
- = assignment
- Perform the indicated operation between the left and right operands, then assign the result to the left operand
  - += add to
  - -= subtract from
  - *= multiply by
  - /= divide by
  - %= modulo by

Example 1: `a=x=y+3`; so `a = x`, right?
Example 2: `r=s+(t=u-v)/3`; give “same as” code.

NOTE: using an assignment operator (=) is legal anywhere it is legal to compare for equality (==), so it is not a syntax error (depends on compiler; may give a warning) because there is not a distinct boolean type in C.
Unary Operators: ++ -- also + -

++a and a++ are both the same as a = a + 1
--a and a-- are both the same as a = a – 1

HOWEVER...
++a → a incremented BEFORE a is used
--a → a decremented BEFORE a is used
a++ → a is incremented AFTER a has been used
a-- → a is decremented AFTER a has been used

In both examples, the final value of a will be 2, BUT...

```c
int main()
{
    int a = 1;
    printf(" a is %d", ++a)
    return 0;
}
/* 2 will be printed */
```

```c
int main()
{
    int a = 1;
    printf(" a is %d", a++)
    return 0;
}
/* 1 will be printed */
```
Logical operators (cont)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Remarks</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;&lt;</code></td>
<td>left-shift</td>
<td>wait for these... later</td>
<td></td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>right-shift</td>
<td>wait for these... later</td>
<td></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>bitwise AND</td>
<td>BITWISE OPERATORS:</td>
<td>0 is FALSE</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>bitwise OR</td>
<td>1 is TRUE</td>
</tr>
<tr>
<td><code>^</code></td>
<td>bitwise exclusive-OR</td>
<td>Result: depends</td>
<td></td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>logical AND</td>
<td>LOGICAL OPERATORS:</td>
<td>0 is FALSE</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
<td>logical OR</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
<td></td>
<td>Result: 0 if false; 1 if true</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>==</code></td>
<td>equals</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>!=</code></td>
<td>does not equal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: C has no keywords as `true` or `false` 'OR', 'AND', and 'NOT': differences between bitwise and logical operators.
Relational Operators

- Used to **compare** two values
- `<  <=  >  >=
- `==  !=`
- Precedence order given above; then left to right
- “else” equivalences (respectively)
  - `>=  >  <=  <`
  - `!=  ==`
- Arithmetic operators have higher precedence than relational operators
- A true statement is one that evaluates to a nonzero number. A false statement evaluates to zero. When you perform comparison with the relational operators, the operator will return 1 if the comparison is true, or 0 if the comparison is false.
  - For example, the check `0 == 2` evaluates to 0. The check `2 == 2` evaluates to a 1.

TRY: `printf("%d",2==1);`
Bitwise operators

- `&` and `&=`: bitwise AND assignment
- `|` and `|=`: bitwise OR assignment
- `^` and `^=`: bitwise exclusive OR assignment
- `<<` and `<<=`: left shift assignment
- `>>` and `>>=`: right shift assignment

Assignment operators take the value on the right and place it into the variable on the left. C provides more derivatives by combining computation and assignment together.

- `~`: inverse each individual bit (one’s complement)

**EXAMPLES**
- `i &= 1 \Rightarrow i = i & 1`
- `i <<= 1 \Rightarrow i = i << 1`
Boolean Operators

- C does not have a distinct boolean type
  - int is used instead

- Treats integer 0 as FALSE and all non-zero values as TRUE
  - i = 0;
  - while (i - 10) {
    - ... }
  - will execute until the variable i takes on the value 10 at which time the expression (i - 10) will become false (i.e. 0).

- A sampling of Logical/Boolean Operators:
  - &&, ||, and ! ➔ AND, OR, and NOT

- For example, && is used to compare two objects
  - x != 0 && y != 0

- Short-Circuit Evaluation: In the above example, if x != 0 evaluates to false, the whole statement is false regardless of the outcome of y != 0
  - same for OR if first condition is true
The boolean operators function in a similar way to the comparison (relational) operators: each returns 0 if evaluates to FALSE or 1 if it evaluates to TRUE.

**NOT**: The NOT operator accepts one input. If that input is TRUE, it returns FALSE, and if that input is FALSE, it returns TRUE. For example, NOT (1) evaluates to 0, and NOT (0) evaluates to 1. NOT (any number but zero) evaluates to 0. In C, NOT is written as !

**AND**: This is another important command. AND returns TRUE if both inputs are TRUE (if 'this' AND 'that' are true). (1) AND (0) would evaluate to zero because one of the inputs is false (both must be TRUE for it to evaluate to TRUE). (1) AND (1) evaluates to 1. (any number but 0) AND (0) evaluates to 0. The AND operator is written && in C. Do not be confused by thinking it checks equality between numbers: it does not.

**OR**: Very useful is the OR statement! If either (or both) of the two values it checks are TRUE then it returns TRUE. For example, (1) OR (0) evaluates to 1. (0) OR (0) evaluates to 0. The OR is written as || in C. Those are the pipe characters. On your keyboard, they may look like a stretched colon. On my computer the pipe shares its key with \. A “unary” operator since it only works on one operand.

**PRECEDENCE**

- NOT is evaluated prior to both AND and OR. Also multiple NOTs are evaluated from right to left.
- AND operator is evaluated before the OR operator. Also, multiple ANDs are evaluated from left to right.
- OR will be evaluated after AND. Also, multiple ORs are evaluated from left to right.

AND (&&) : Returns true only if both operand are true.

OR (||) : Returns true if one of the operand is true.

NOT (!) : Converts false to true and true to false.
## Boolean Examples

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator's Name</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>3&gt;2 &amp;&amp; 3&gt;1</td>
<td>1(true)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>3&gt;2 &amp;&amp; 3&lt;1</td>
<td>0(false)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td>3&lt;2 &amp;&amp; 3&lt;1</td>
<td>0(false)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td>!(3==2)</td>
<td>1(true)</td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td>!(3==3)</td>
<td>0(false)</td>
</tr>
</tbody>
</table>

**A.** !(1 || 0)  
**ANSWER:** 0

**B.** !(1 || 1 && 0)  
**ANSWER:** 0 (AND is evaluated before OR)

**C.** !( (1 || 0) && 0 )  
**ANSWER:** 1 (Parenthesis are useful)
In-Class Exercises

```c
#include <stdio.h>    // test.c
main() {
    int z, a=10, b=0, c=3;
    z = (a > c) || (++a > b);
    printf("z = %d  a = %d\n", z, a);
    c = 30;
    z = (a > c) || (++a > b);
    printf("z = %d  a = %d\n", z, a);
    z = ! ~ b++ ;
    printf("z = %d  b = %d\n", z, b);
    a = 10, c=3;
    z = ++a * c + a++;
    printf("z = %d  a = %d", z, a);
    // getchar();
    return 0;
}
```
In-Class Exercise

Years that are divisible by four are leap years with one exception – years that are divisible by 100 are not. However, years that are divisible by 400 are leap years. Write a single assignment that sets the identifier leap-year to true if the value in year is a leap year and false if it is not.
### Arithmetic type issues

- **Type casting - EXPLICIT**
  - A way to convert a variable from one data type to another data type
  - Uses a cast operator → (type-name)
  - (type-name) expression

- **Type combination and promotion - IMPLICIT**
  - ('a' – 32) = 97 – 32 = 65 = ‘A’
  - Smaller type (char) is “promoted” to be the same size as the larger type (int)
  - Determined at compile time - based purely on the *types* of the values in the expressions
  - Does not lose information – convert from type to compatible large type
The **usual arithmetic conversions** are implicitly performed to cast their values in a common type. Compiler first performs *integer promotion*, if operands still have different types then they are converted to the type that appears highest in the following hierarchy →

```
long double
  ↑
double
  ↑
float
  ↑
unsigned long long
  ↑
long long
  ↑
unsigned long
  ↑
long
  ↑
unsigned int
  ↑
int
```
Arithmetic type conversions (fyi)

Usual Arithmetic Conversions ➔ Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*.

- If either operand is long double, the other is converted to long double.
- If either operand is double, the other is converted to double.
- If either operand is float, the other is converted to float.

Otherwise, the integral promotions are performed on both operands;

- If either operand is unsigned long int, the other is converted to unsigned long int.
- If one operand is long int and the other is unsigned int, the effect depends on whether a long int can represent all values of an unsigned int; if so, the unsigned int operand is converted to long int; if not, both are converted to unsigned long int.
- If one operand is long int, the other is converted to long int.
- If either operand is unsigned int, the other is converted to unsigned int.

Otherwise, both operands have type int.

NOTE: There are two changes here. First, arithmetic on float operands may be done in single precision, rather than double; the first edition specified that all floating arithmetic was double precision. Second, shorter unsigned types, when combined with a larger signed type, do not propagate the unsigned property to the result type; in the first edition, the unsigned always dominated. The new rules are slightly more complicated, but reduce somewhat the surprises that may occur when an unsigned quantity meets signed. Unexpected results may still occur when an unsigned expression is compared to a signed expression of the same size.
In-Class Exercises

The following code is supposed to scale a homework score in the range 0..20 to be in the range 0..100.

```c
int score;
// score given a value here
score = (score / 20) * 100;
```

What about...

```c
score = ((double) score / 20) * 100;
score = (score / 20.0) * 100;
score = (int) (score / 20.0) * 100;
```

What is the type and value of the expression `(float) (25/10)`?
Basic I/O Example

#include <stdio.h>

int main()
{
    int first, second, add;
    float divide;

    printf("Enter two integers\n");
    scanf("%d %d", &first, &second);

    add = first + second;
    divide = first / (float) second;

    printf("Sum = %d\n", add);
    printf("Division = %.2f\n", divide);

    return 0;
}

/* basicio.c */
Basic I/O

- There is no input or output defined in C itself
- Character based (no format specifiers) – character by character I/O
  - getchar() - input
  - putchar(c) - output
- Formatted - standard I/O
  - scanf(stuff goes in here) - input *** white space is important!!
  - printf(stuff goes in here) - output
  - Format Specifiers (% before specifier) – see next slide

```c
#include <stdio.h>
int main(void) {
    int i = 65; /* what if 258 instead of 65? */
    char a;
    printf("i=n",i);
    printf("output with a putchar ");
    putchar(i);
    a = (char) i;
    printf("a=n",a);
    getchar();
    return(0); } /* check.c */
```

```c
#include <stdio.h>
int main() { /* check1.c */
    int x;
    scanf("%d\n", &x);
    printf("x=%d\n", x);
}
```

Q. Why are pointers given to scanf?
A. Needs a pointer to the variable if it is going to change the variable itself i.e. assign a value to x.
When programming in C, you use conversion characters — the percent sign and a letter, for the most part — as placeholders for variables you want to display. The following table shows the conversion characters and what they display:

<table>
<thead>
<tr>
<th>Conversion Character</th>
<th>Displays Argument (Variable’s Contents) As</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>Single character</td>
</tr>
<tr>
<td>%d, %i</td>
<td>Signed decimal integer (int)</td>
</tr>
<tr>
<td>%e</td>
<td>Signed floating-point value in E notation</td>
</tr>
<tr>
<td>%f</td>
<td>Signed floating-point value (float)</td>
</tr>
<tr>
<td>%g</td>
<td>Signed value in %e or %f format, whichever is shorter</td>
</tr>
<tr>
<td>%o</td>
<td>Unsigned octal (base 8) integer (int)</td>
</tr>
<tr>
<td>%s</td>
<td>String of text</td>
</tr>
<tr>
<td>%u</td>
<td>Unsigned decimal integer (int)</td>
</tr>
<tr>
<td>%x</td>
<td>Unsigned hexadecimal (base 16) integer (int)</td>
</tr>
<tr>
<td>%%</td>
<td>(percent character)</td>
</tr>
</tbody>
</table>
Printf formatted output conversions

<table>
<thead>
<tr>
<th>Character</th>
<th>Argument type; Printed As</th>
</tr>
</thead>
<tbody>
<tr>
<td>d,i</td>
<td>int; signed decimal notation.</td>
</tr>
<tr>
<td>o</td>
<td>int; unsigned octal notation (without a leading zero).</td>
</tr>
<tr>
<td>x,X</td>
<td>unsigned int; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef for 0x or ABCDEF for 0X.</td>
</tr>
<tr>
<td>u</td>
<td>int; unsigned decimal notation.</td>
</tr>
<tr>
<td>c</td>
<td>int; single character, after conversion to unsigned char ***</td>
</tr>
<tr>
<td>s</td>
<td>characters from the string are printed until a \’\0\’ is reached or until the number of characters indicated by the precision have been printed.</td>
</tr>
<tr>
<td>f</td>
<td>double; decimal notation of the form [-]mmm.ddd, where the number of d’s is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.</td>
</tr>
<tr>
<td>e,E</td>
<td>double; decimal notation of the form [-]m.ddddde+/xx or [-]m.ddddde+/xx, where the number of d’s is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.</td>
</tr>
<tr>
<td>g,G</td>
<td>double; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal point are not printed.</td>
</tr>
</tbody>
</table>
Decimal & Floating point

- **Width of the whole number portion**
  - For decimal integers (works with %i as well)
- The character width for float
  - includes the decimal point position

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>print as decimal integer</td>
</tr>
<tr>
<td>%6d</td>
<td>print as decimal integer, at least 6 characters wide</td>
</tr>
<tr>
<td>%f</td>
<td>print as floating point</td>
</tr>
<tr>
<td>%6f</td>
<td>print as floating point, at least 6 characters wide</td>
</tr>
<tr>
<td>%.2f</td>
<td>print as floating point, 2 characters after decimal point</td>
</tr>
<tr>
<td>%6.2f</td>
<td>print as floating point, at least 6 wide and 2 after decimal point</td>
</tr>
</tbody>
</table>
Printf examples

causes the values of the two integers fahr and celsius to be printed, with a tab (\t) between them

printf("%d\t%d\n", fahr, celsius);

to print the first number of each line in a field three digits wide, and the second in a field six digits wide

printf("%3d %6d\n", fahr, celsius);

Each % construction in the first argument of printf is paired with the corresponding second argument, third argument, etc.; they must match up properly by number and type, or you will get wrong answers.

printf("\na=%f\nb=%f\nc=%f\nd=%f", a, b, c, d);
c = a + b;
printf("%d + %d = %d\n", a, b, c);
Printf and Scanf

- Both formatted I/O
- Both sent to “standard I/O” location

Printf

- Converts values to character form according to the format string

Scanf

- Converts characters according to the format string, and followed by pointer arguments indicating where the resulting values are stored
Scanf requires two inputs:
- String argument - with format specifiers
- Set of additional arguments (pointers to variables)

Consists of % at the beginning and a type indicator at the end
Skips over all leading white space (spaces, tabs, and newlines) prior to finding first input value
In between options:
- * = used to suppress input
- maximum field-width indicator
- type indicator modifier
Input stops when:
- End of format string
- Input read does not match what the format string specifies i.e. pointer arguments MUST BE the right type
- The next call to scanf resumes searching immediately after the last character already converted.

Return value = # of values converted

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>MEANING</th>
<th>VARIABLE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>read an integer value</td>
<td>int</td>
</tr>
<tr>
<td>%ld</td>
<td>read a long integer value</td>
<td>long</td>
</tr>
<tr>
<td>%f</td>
<td>read a real value</td>
<td>float</td>
</tr>
<tr>
<td>%lf</td>
<td>read a double precision real value</td>
<td>double</td>
</tr>
<tr>
<td>%c</td>
<td>read a character</td>
<td>char</td>
</tr>
<tr>
<td>%s</td>
<td>read a character string from the input</td>
<td>array of char</td>
</tr>
</tbody>
</table>
**Scanf examples**

```c
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year);
Input:
01/29/64

int anInt;
scanf("%i%%", &anInt);
Input:
23%
anInt==23

int anInt; long l;
scanf("%d %ld", &anInt, &l);
Input:
-23 200
anInt==23
l==200

NOTE: You need to follow each `scanf` with a `getchar` to throw away the newline character!
```

```c
double d;
scanf("%lf", &d);
Input:
3.14
d==3.14

int anInt;
scanf("%*s %i", &anInt);
Input:
Age: 29
anInt==29 result

string s;
scanf("%9s", s);
Input:
VeryLongString
s==“VeryLongS”
Why no ‘&’ before s?
```

```c
int anInt, anInt2;
scanf("%2i", &anInt);
scanf("%2i", &anInt2);
Input:
2345
anInt==23
anInt2==45
```

**NOTE:** pressing the enter key means you have entered a character...
## Scapf practice problems

<table>
<thead>
<tr>
<th>Letter</th>
<th>Type of Matching Argument</th>
<th>Auto-skip; Leading White-Space</th>
<th>Example</th>
<th>Sample Matching Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>% (a literal, matched but not converted or assigned)</td>
<td>no</td>
<td>int anInt; scanf(&quot;%i%%&quot;, &amp;anInt);</td>
<td>23%</td>
</tr>
<tr>
<td>d</td>
<td>int</td>
<td>yes</td>
<td>int anInt; long l; scanf(&quot;%d %ld&quot;, &amp;anInt, &amp;l);</td>
<td>-23 200</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>yes</td>
<td>int anInt; scanf(&quot;%i&quot;, &amp;anInt);</td>
<td>0x23</td>
</tr>
<tr>
<td>o</td>
<td>unsigned int</td>
<td>yes</td>
<td>unsigned int aUInt; scanf(&quot;%o&quot;, &amp;aUInt);</td>
<td>023</td>
</tr>
<tr>
<td>u</td>
<td>unsigned int</td>
<td>yes</td>
<td>unsigned int aUInt; scanf(&quot;%u&quot;, &amp;aUInt);</td>
<td>23</td>
</tr>
<tr>
<td>x</td>
<td>unsigned int</td>
<td>yes</td>
<td>unsigned int aUInt; scanf(&quot;%x&quot;, &amp;aUInt);</td>
<td>1A</td>
</tr>
<tr>
<td>a, e, f, g</td>
<td>float or double</td>
<td>yes</td>
<td>float f; double d; scanf(&quot;%f %lf&quot;, &amp;f, &amp;d);</td>
<td>1.2 3.4</td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>no</td>
<td>char ch; scanf(&quot; %c&quot;, &amp;ch);</td>
<td>Q</td>
</tr>
<tr>
<td>s</td>
<td>array of char</td>
<td>yes</td>
<td>char s[30]; scanf(&quot;%29s&quot;, s);</td>
<td>hello</td>
</tr>
<tr>
<td>n</td>
<td>int</td>
<td>no</td>
<td>int x, cnt; scanf(&quot;X: %d%n&quot;, &amp;x, &amp;cnt);</td>
<td>X: 123 (cnt==6)</td>
</tr>
<tr>
<td>[</td>
<td>array of char</td>
<td>no</td>
<td>char s1[64], s2[64]; scanf(&quot;%[^\n]%[^\n]&quot;, s1); scanf(&quot;%[^\t] %[^\t]&quot;, s1, s2);</td>
<td>Hello World field1 field2</td>
</tr>
</tbody>
</table>
You can use this function even with spaces in the input:

```
scanf(" %[^\n]s",a);
```

Explanation: Converts the longest string consisting of only the characters after the ']' up to the first matching '['. If the first character after the '[' is '^', then the matching string is all characters up to the first of the listed characters. (A ']’ character can be included in the list if it is first "%[]ABC]" or "%^[ABC]". A hyphen must be first or last to be included ("%[ABC-]"). A hyphen in the middle indicates a range ("%[A-Za-z0-9]"), although this is implementation defined. This conversion is often used to read a line up to a newline, or fields from (tab) delimited data files as in the example. The resulting string is null-terminated.
## Loop constructs

<table>
<thead>
<tr>
<th>Loop Construct</th>
<th>Description</th>
</tr>
</thead>
</table>
| **for** (init; cond; modify) { statement(s); } | - **Statement(s) only execute when the condition is TRUE.**  
- Notice the semi-colon locations.  
- Do/while always executes at least once (not necessarily true for the other two constructs).  
- For loop → initialize a value, test the condition, if condition is true, execute the statements (if condition is false, exit the loop), modify the value, test the condition, if the condition is true, execute the statements, etc. |
| **while** (cond) { statement(s); } | |
| **do** { statement(s); } **while** (cond); | |

```c
#include <stdio.h>

int main () {
    int n, sum;
    sum = 0;
    for (n = 1; n <= 10; n=n+1) // n++ also n+=1 also ++n
    {
        sum = sum + n;   // technically, do not need the brackets
    }
    printf("\n The sum of integers from 1 to 10 is %d, have a nice day", sum);
    return 0;
}
```

```c
n=1;
while (n<=10) {
    sum=sum+n;
    n=n+1;
}
```

```c
n=0;
do {
    n=n+1;
    sum=sum+n;
} while (n<10);
```
Loop construct examples

```c
#include <stdio.h>
int main()
{
    int x;
    x = 0;
    do {
        printf( "Hello, world!\n" );
    } while ( x != 0 );
    getchar();
    return 0;
}
/* "Hello, world!" is printed at least one time even though the condition is false */
```

```c
#include <stdio.h>
int main()
{
    int x = 0;
    /* Don't forget to declare variables */
    while ( x < 10 ) {
        /* While x is less than 10 */
        printf( "%d\n", x );
        x++;
    }
    /* Update x so the condition can be met eventually */
    getchar();
    return 0;
}
```

```c
#include <stdio.h>
int main()
{
    int x;
    for ( x = 0; x < 10; x++ ) {
        printf( "%d\n", x );
    }
    getchar();
    return 0;
}
/* The loop goes while x < 10, and x increases by one every loop */
```
Infinite Loops

#include <stdio.h>
#define TRUE 1
int main()
{
    while (TRUE)
    {
        printf("Hello World\n");
        return 0;
    }

QUESTIONS:
1. What if indent the return statement?
2. Create a variable to replace the while condition.
3. Put brackets around the while statements.
Keywords that are very important to looping are **break** and **continue**.

**BREAK** command will
- exit the most immediately surrounding loop regardless of what the conditions of the loop are.
- Break is useful if we want to exit a loop under special circumstances.

**CONTINUE** is another keyword that controls the flow of loops. If you are executing a loop and hit a continue statement, the loop will stop its current iteration, update itself (in the case of FOR loops) and begin to execute again from the top. Essentially, the continue statement is saying "this iteration of the loop is done, let's continue with the loop without executing whatever code comes after me."

GOTO next slide (really, continued on the next slide)
Jump Statement

- **Goto plus a labeled statement**
  - `goto identifier;`
  - `identifier: statement;`

- Have to declare the identifier???
  - NO!

- A statement label is meaningful only to a **goto** statement; in any other context, a labeled statement is executed without regard to the label (i.e. is ignored).

- A **jump-statement** must reside in the same function and can appear before only one statement in the same function.
- The set of **identifier** names following a **goto** has its own name space so the names do not interfere with other identifiers.
- Labels cannot be redeclared.

- It is good programming style to use the **break**, **continue**, and **return** statement in preference to **goto** whenever possible. However, since the **break** statement only exits from one level of the loop, a **goto** may be necessary for exiting a loop from within a deeply nested loop.
while (true) {
    if (someone_has_won() || someone_wants_to_quit() == TRUE) {
        break;
    }
    take_turn(player1);
    if (someone_has_won() || someone_wants_to_quit() == TRUE) {
        break;
    }
    take_turn(player2); }
    /* playing checkers */

for (player = 1; someone_has_won == FALSE; player++) {
    if (player > total_number_of_players) {
        player = 1;
    }
    if (is_bankrupt(player)) {
        continue;
    }
    take_turn(player); }  /* playing monopoly */
Goto example

#include <stdio.h>
int main() {
    int i, j;
    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 3; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 5 )
                goto stop;
        }
    }
    /* This message does not print: */
    printf_s( "Loop exited. i = %d\n", i );
stop: printf( "Jumped to stop. i = %d\n", i );
return 0; }

In this example, a goto statement transfers control to the point labeled stop when i equals 5.
If, else-if, switch-case conditional statements

```java
if ( TRUE ) {
    /* Execute these stmts if TRUE */
} else {
    /* Execute these stmts if FALSE */
}
```

```java
if (condition) {
    statement(s); }
else if (condition) {
    statement(s); }
else {
    statement(s); }
```

```java
switch ( <variable> ) {
    case this-value:    /* Note the :, not a ; */
        Code to execute if <variable> == this-value;
        break;
    case that-value:
        Code to execute if <variable> == that-value;
        break;
    ... default:
        Code to execute if <variable> does not equal the value following any of the cases... break; }
```

SWITCH NOTES:
- Notice, no {} blocks within each case.
- Notice the colon for each case and value.
- The “condition” of a switch statement is a value.
- The default case is optional, but it is wise to include it as it handles any unexpected cases.
- Chooses first match...
**ElseIF example**

```c
#include <stdio.h>
int main()  {
    int age;
    printf( "Please enter your age" );
    scanf( "%d", &age );
    if ( age < 100 ) {
        printf("You are pretty young!\n");
    } else if ( age == 100 ) {
        printf( "You are old\n" );
    } else {
        printf( "You are really old\n" );
    }
    return 0;
}
```

NOTE: You do not have to use {} if only one statement in the block. None of the above brackets in the IF structure are necessary! Check out where the semi-colon goes (and where it doesn’t).
The switch statement is a way to execute different blocks of code based on a certain condition. In the example given, the condition is the variable `x`. The switch statement in C is:

```c
switch ( x ) {
    case 'a':
        /* Do stuff when x is 'a' */
        break;
    case 'b':
    case 'c':
    case 'd':
        /* Fallthrough technique...
           cases b,c,d all use this code */
        break;
    default:
        /* Handle cases when x is not a,b,c or d. ALWAYS have a default case*/
        break;
}
```

The fallthrough technique allows the execution of the code after the break statement to continue without the need for an explicit case for matching `x`. If the default case is not present, the program will not handle cases that do not match any of the defined cases.

To demonstrate the use of a switch statement, consider the following program:

```c
#include <stdio.h>

void playgame() { printf( "Play game called" ); }
void loadgame() { printf( "Load game called" ); }
void playmultiplayer() { printf( "Play multiplayer game called" ); }
int main() {
    int input;
    printf( "1. Play game\n" );
    printf( "2. Load game\n" );
    printf( "3. Play multiplayer\n" );
    printf( "4. Exit\n" );
    printf( "Selection: " );
    scanf( "%d", &input );
    switch ( input ) {
        case 1:
            playgame();
            break;
        case 2:
            loadgame();
            break;
        case 3:
            playmultiplayer();
            break;
        case 4:
            printf( "Thanks for playing!\n" );
            break;
        default:
            printf( "Bad input, quitting!\n" );
            break;
    }
    getchar();
    return 0; }
```
More about BOOLEAN issues

- Every boolean test is an implicit comparison against zero (0).
- However, zero is not a simple concept. It represents:
  - the integer zero for all integral types
  - the floating point 0.0 (positive or negative)
  - the nul character ("0")
  - the null pointer
- In order to make your intentions clear, explicitly show the comparison with zero for all scalars, floating-point numbers, and characters.

Write an INFINITE LOOP as:

- for (;;) ...
- while (1) ... // while (true)

The former is idiomatic among C programmers, and is more visually distinctive.

```c
int i; if (i) is better seen as if (i != 0)
float x; if (!x) is better seen as if (x == 0.0)
char c; if (c) is better seen as (c != '0')
```

An exception is made for pointers, since 0 is the only language-level representation for the null pointer.

```c
/* The symbol NULL is not part of the core language - you have to include a special header file to get it defined */
```

In short, pretend that C has an actual boolean type which is returned by the logical operators and expected by the test constructs, and pretend that the null pointer is a synonym for false.
Conditional Operator

- Consists of two symbols
  - Question mark
  - Colon

- Syntax: `exp1 ? exp2 : exp3`

- Evaluation:
  - If `exp1` is true, then `exp2` is the resulting value
  - If `exp1` is false, then `exp3` is the resulting value

- Example: if `a = 10` and `b = 15`
  - `x = (a > b) ? a : b`
  - `b` is the resulting value and assigned to `x`
  - Parentheses not necessary
  - Similar, but shorter than, if/else statement
The Comma Operator

- Used to link related expressions together
- Evaluated from left to right
- The value of the right most expression is the value of the combined expression
- Example:
  - value = (x = 10, y = 5, x + y);
  - Comma operator has lowest precedence
    - Parentheses are necessary!
  - For loop:
    - for (n=1, m=10; n<=m; n++, m--)
  - While:
    - while (c=getchar(), c!= ‘1’)
  - Exchanging values:
    - t=x, x=y, y=t;

<table>
<thead>
<tr>
<th>with ()</th>
<th>without ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=10;</td>
<td>x=10;</td>
</tr>
<tr>
<td>y=5;</td>
<td>value = x;</td>
</tr>
<tr>
<td>value = x+y;</td>
<td>y=5</td>
</tr>
<tr>
<td></td>
<td>x+y is not stored</td>
</tr>
</tbody>
</table>

Initialize n=1 and m=10
Test
Execute statements
Modify: add 1 to n, subtract 1 from m
Test, modify, etc, until test is false
n/m → 1/10, 2/9, 3/8, 4/7, 5/6, 6/5 exit
An enumerated type is:
- one whose values are symbolic constants rather than literal
- used to define variables with certain discrete integer values
- vs Basic Data types?
- vs #define?

**Declaration**

- example: // defining a type
  ```
  enum Jar_Type {CUP, PINT, QUART, HALF_GALLON, GALLON};
  ```
  declares a *type* called Jar_Type
  - CUP, PINT, etc are symbolic constants
  - {} values are associated with integer values which start at 0 by default
    - CUP=0; PINT=1; QUART=2; HALF_GALLON=3; GALLON=4;

**Variables**

- of this type are declared like this:
  ```
  enum Jar_Type milk_jug, gas_can, medicine_bottle;
  ```

**Other options:**
- ```
  enum { CUP, PINT, QUARTER, HALF_GALLON, GALLON} milk_jug, gas_can, medicine_bottle; // no actual enum type
  ```
- ```
  enum Jar_Type {CUP, PINT, QUART, HALF_GALLON, GALLON } soda;
  ```
Internally, the symbolic names are treated as integer constants which you can assign yourself; even including duplicate values.

See examples (next slide)

Caution: don’t mix them indiscriminately with integers – even though it is viable – weakens the value of enumerated types.

- milk_jug = -623; // variable of type Jar_Type
- int a = PINT; // integer symbolic constant
- PINT = 50; // ILLEGAL
// what are the values for monday, tuesday, etc?
enum DAY { saturday, sunday = 0, monday, tuesday, wednesday, thursday, friday } workday;

// declare another variable, today, of enum type DAY
// and assign the variable an initial value
enum DAY today = wednesday;

// type casting... but not necessary
workday = ( enum DAY ) ( day_value + 1 );

enum MONTH { jan = 1, feb=2, mar=3, apr=4, may=5, jun=6, jul=7, aug=8, sep=9,
            oct=10, nov=11, dec=12 } this_month; // only have to define jan=1
this_month = feb;
printf("What month is it? %d\n",this_month);
More examples, hmmm....

```c
enum BOOLEAN { false, true }; // false = 0, true = 1
enum BOOLEAN end_flag, match_flag;

OR

define BOOLEAN { false, true } end_flag, match_flag;

OR

define BOOLEAN { false, true } end_flag;
define BOOLEAN match_flag;

OR

OR

define { false, true } end_flag, match_flag;
// unnamed enumerated type
```

```c
if ( match_flag == false )
    { ... statement(s) ... }

OR

define_flag = true;
```
Constant Variables

- The values of some variable may be required to remain constant throughout the program.
  - using the qualifier `const` at the time of initialization in 2 ways:
    - `const int size = 40;`
    - `int const size = 40;`

- The `const` data type qualifier tells the compiler that the value of the `int` variable `size` may not be modified in the program.

- Assignment
  - At declaration
  - During function call for `const` parameters

- Different from `#define` — both creating named constants
  - `#define MAX_ELEMENTS 50 // literal constant`
  - `int const max_elements = 50; // variable constant`
  - Constant variable can only be used where variables are allowed; literal constant is allowed wherever a constant is allowed, such as in declaring the size of arrays.
What is GDB?

- **GDB: The GNU Project Debugger**
  - Allows you to see what is going on “inside” another program while it executes -- or what another program was doing at the moment it crashed.
  - GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act*:
    - Start your program, specifying anything that might affect its behavior.
    - Make your program stop on specified conditions.
    - Examine what has happened, when your program has stopped.
    - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

- or just for fun to see what is going on behind the scenes :o)
- [http://www.tutorialspoint.com/gnu_debugger/gdb_debugging_programs.htm](http://www.tutorialspoint.com/gnu_debugger/gdb_debugging_programs.htm)
Using GDB

- %nl gdbincl.c > gdbinclnl  // number lines of files
- gdbtestnl is a text file so no extension necessary
- Use an editor to open gdbinclnl
- Now can reference line numbers

- %more gdbincl.c
  - Shows your program on the screen

- Need to compile with –g option
  - gcc –g –o hello hello.c

- Run using %gdb exec-filename

COMMANDS/tutorial

- help – lists gdb command topics
- info xxx – where xxx be to list the breakpoints, breakpoint numbers, registers, etc
- run – starts execution
- quit – short cut is just q

GDB command (cont)

- **Break and watch commands**
  - break/tbreak followed by:
    - Function name, line number
  - clear – delete breakpoints
  - watch – followed by a condition
    - Suspends processing when condition is met
  - delete – delete all break/watch points
  - continue – exec until next break/watch point
  - finish – continue to end of function

- **Line execution commands**
  - step – step to next line of code (will step into a function)
  - next – execute next line of code (will not enter functions)
  - until - continue processing until you reach a specified line number
Redirection File I/O

- Part of the operating system (linux)
- `% lab1p < lab1p1in >! lab1p1out`
  - `<` gets input from a file instead of the keyboard
  - `>` redirects output to a file instead of the screen/monitor
  - `!` overwrites if the file already exists

```c
while (TRUE) {
    chdigit=getchar();
    while (chdigit != '\n')
    { // loop stuff...
        chdigit=getchar();
    }
    if (numvals==1 && intnum=0)
        break;
    else
        // output average of one line
}
```

**Input File:**
```
92 56 78 100 82
100 18 45 66
88 99 7
82
9
78 100 45 32 87 69 55
0
```
C built-in Functions


In-Class Exercise

- How do you find C functions?
- What does printf return?
- Give one function in the math.h file
A look behind the scenes...

<table>
<thead>
<tr>
<th>MEMORY - Programmer Address Space</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>high memory address</td>
<td></td>
</tr>
<tr>
<td>other</td>
<td></td>
</tr>
<tr>
<td>STACK</td>
<td></td>
</tr>
<tr>
<td>local variables stored here</td>
<td></td>
</tr>
<tr>
<td>... (auto)</td>
<td></td>
</tr>
<tr>
<td>HEAP</td>
<td></td>
</tr>
<tr>
<td>pointers managed by programmer</td>
<td></td>
</tr>
<tr>
<td>static variables stored here (and global)</td>
<td></td>
</tr>
<tr>
<td>constant variables stored here</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>program instructions</td>
<td></td>
</tr>
<tr>
<td>low memory address</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Declaring variables

- SCOPE = A region of a program where a defined variable can have its existence and beyond that, the variable can not be accessed.

- 3 places where variables can be declared:
  - Inside a function or block ➔ LOCAL variable
  - Outside a function ➔ GLOBAL variable
  - Definition of a function parameter ➔ FORMAL parameter

- THE LOCATION WHERE AN IDENTIFIER IS DECLARED DETERMINES ITS SCOPE
Scope

- An area in the program in which an identifier may be used.
  For example, the scope of a function’s local variables is limited to the body of the function.
- This means:
  - No other function may access these variables by their names because the names are not valid outside of their scope.
  - It is legal to declare different variables with the same names so long as they are not declared in the same scope.

Scope types:
- File
- Function
- Block
- Prototype
Enumerated data types

The identifiers in the enumeration list must be distinct from other identifiers in the same scope with the same visibility, including ordinary variable names and identifiers in other enumeration lists.

Enumeration tags (type name) obey the normal scoping rules. They must be distinct from other enumeration, structure, and union tags with the same visibility.
A block is a list of statement enclosed in braces
Any identifiers declared at the beginning of the block are accessible to all statements in the block
The formal parameters (not prototypes) of a function definition also have block scope in the function’s body

- Local variables declared in the outermost block cannot have the same name as any of the parameters because they are all declared in the same scope

Nested blocks having declarations of variables with the same name

- The outer block variable cannot be referenced by name from within the inner block (try to avoid)
Any identifier declared outside of all blocks
Means that the identifier may be accessed anywhere from its declaration to the end of the source file in which it was declared
Prototype scope

- Applies only to argument names declared in function prototypes

Reminder:
- argument names need not appear. If given, any names can be chosen as they need not match either the formal parameter names given in the function definition or the names of the actual arguments used when the function is called

- Prevents these name from conflicting with any other names in the program

- Only possible conflict is using the same name more than once in the same prototype
Function scope

- Only applies to statement labels which are used with goto statements
- One simple rule: all statement labels in a function must be unique
  - I hope you never use this knowledge!
Scope Example – In class assignment

```c
int a;
int b (int c);
int d (int e)
{
    int f;
    int g (int h);
    ...
    {
        int f, g, i;
        ...
    }
    {
        int i;
        ...
    }
}
```

1 → int a;
2 → int b  3 → int c
4 → int d  5 → int e

6 → int f
7 → int g  8 → int h

9 → int f, g, i

10 → int i

File scope = 1,2,4
Prototype scope = 3,8
Block = 6, 7, 9, 10  (f?)
5 = block scope in function body
? What if 6 had included an e?
Storage Class

Storage class defines:

- **Scope (visibility)**
  - region of the program where a defined variable can have its existence and beyond that, the variable can not be accessed

- **Life time**
  - determines when the variable is created and destroyed; and how long it will retain its value

of variables and/or functions

The specifiers precede the type they modify

- Auto
- Register
- Static
- Extern – learn more about later
Auto/Register Storage classes

- Auto
  - Default for all local variables
  - Can only be used within functions
  - Two variables with the same storage class
    - int month;
    - auto int month;
  - Stored on the stack

- Register
  - Local variables that should be stored in a register
  - Variable has a maximum size equal to the register size (defined by the hardware)
  - Variable does not have a memory location (no address)
  - Used for variables that are used a lot and require quick access; like a counter
    - register int miles;
  - Might be stored in a register
    - Depends on hardware and implementation restrictions
  - Must be initialized explicitly
Storage classes - register

- Can be used on automatic variables to indicate that they should be stored in the machine’s hardware registers rather than in memory.
- **WHY?** To be accessed more efficiently
- **FYI** – compiler can ignore if necessary i.e. too many designated as `register` (first come first served) → rest are automatic
- Smart Compiler? One that does its own register optimization so may ignore register class altogether and decide for itself. *Skynet, is that you?*
- Typically declare heavily used variables as `register` variables
- Created and destroyed at the same time as automatic variables
- Not allowed to take the address of a register variable
Storage Class

- Static – instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into the goes out of scope, that is:
  - Local variable which exists and retains its value even after the control is transferred to the calling function;
  - automatically initialized to zero;
  - initialized only once during compilation;
  - commonly used along with functions;
  - stored in ordinary memory;
  - only one copy.
Storage Classes – static

- Variables declared *within a block* but with the keyword `static` changes storage class from automatic to static.
- Static storage class exists for the entire duration of the program, rather than just the duration of the block in which it is declared.
- NOTE: the changing of the storage class of a variable does not change its scope; it is still accessible by name only from within the block.
- FYI: formal parameters to a function cannot be declared static, because arguments are always passed on the stack to support recursion.
When used in function definitions, or declarations of variables that appear outside of blocks

- The keyword static changes the linkage from external to internal*
- The storage class and scope are not affected
- Accessible only from within the source file in which they were declared

* Explain when go over linkage
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {
    while(count--)
        func();
    return 0;
}

/* function definition */
void func( void ) {
    static int i = 5; /* local static variable */
    i++;
    printf("i is %d and count is %d\n", i, count);
}

OUTPUT:
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
The default storage class for a variable depends on where it is declared... inside a block or outside a block.
Storage Classes - blocks

- **Outside any blocks**
  - Always stored in static memory (ordinary memory)
  - No way to specify any other storage class for these variables
  - Static variables are created before the program begins to run and exist throughout its entire execution.
  - They retain whatever value they were assigned until a different value is assigned or until the program completes

- **Within a block**
  - Default storage class is automatic
  - Stored on the stack
  - Keyword auto rarely used because doesn’t change default
  - Created just before the program execution enters the block in which they are declared;
  - Discarded just as execution leaves that block
  - New copies created each time the block is executed
## Identifier Storage Class Summary

- **Automatic**
  - Default
  - Local to a block
  - Discarded on exit from block
  - Can have `auto` specifier
  - Can have `register` specifier
    - Stored in fast registers of the machine if possible instead of RAM

- **Static**
  - Default for global variables
    - Declared prior to the main() function
  - Can also be defined within a function
  - Initialized at compile time and retains its value between calls... initial value must be a constant... be careful!
Example

Suppose you want to write two functions, x and y, in the same source file, that use the variables given below. How and where would you write the declarations? NOTE: all initializations must be made in the declarations themselves, not by any executable statements in the functions.

The trick is to realize that function y can be put ahead of x in the file; after that, the rest is straightforward. Watch for assignment statements though; the problem specifies no executable statements in the functions.

```
static char b = 2;
void y( void )
{
}
int a = 1;

void x( void )
{
int c = 3;
static float d = 4;
}
```

<table>
<thead>
<tr>
<th>Nm/Ty</th>
<th>STORAGE</th>
<th>LINKAGE</th>
<th>SCOPE &amp; INITIAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = int</td>
<td>static</td>
<td>external</td>
<td>accessible to x but not y with init value = 1</td>
</tr>
<tr>
<td>b = char</td>
<td>static</td>
<td>none</td>
<td>accessible to x and y with init value = 2</td>
</tr>
<tr>
<td>c = int</td>
<td>automatic</td>
<td>none</td>
<td>local to x with init value = 3</td>
</tr>
<tr>
<td>d = float</td>
<td>static</td>
<td>none</td>
<td>local to x with init value = 4</td>
</tr>
<tr>
<td>Variable Type</td>
<td>Where Declared</td>
<td>Stored on Stack</td>
<td>Scope</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td>-----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>global</td>
<td>outside of all blocks</td>
<td>no (1)</td>
<td>remainder of this source file</td>
</tr>
<tr>
<td>local</td>
<td>beginning of a block</td>
<td>yes (2)</td>
<td>throughout the block (3)</td>
</tr>
<tr>
<td>formal parameters</td>
<td>function header</td>
<td>yes (2)</td>
<td>throughout the function (3)</td>
</tr>
</tbody>
</table>

(1) variables stored on the stack retain their values only while the block to which they are local is active. When execution leaves the block, the values are lost.

(2) Variables *not* stored on the stack are created when the program begins executing and retain their values throughout execution, regardless of whether they are local or global.

(3) except in nested blocks that declare identical names.
Linkage

- After the individual source files comprising a program are compiled, the object files are linked together with functions from one or more libraries to form the executable program.

- When the same identifier appears in more than one source file, do they refer to the same entity or two different entities???
## Linkage Types

- Determines how multiple occurrences of an identifier are treated
- The scope of an identifier is related to its linkage, but the two properties are not the same
- **3 types**
  - **None**
    - Identifiers that have no linkage are always individuals i.e. multiple declarations of the same identifier are always treated as separate and distinct entities
  - **Internal**
    - All declarations of the identifier within one source file refer to a single entity, but declarations of the same identifier in other source files refer to different entities
  - **External**
    - All references to an identifier refer to the same entity
    - Global variable known to all functions in the file; declared outside the main() function; automatically initialized to zero.
External example

First File: main.c
#include <stdio.h>
int count;
extern void write_extern();
main() {
    count = 5;
    write_extern();
}

Second File: write.c
#include <stdio.h>
extern int count;
void write_extern(void) {
    printf("count is %d\n", count);
} // outputs → count is 5

Don’t initialize here

Declarations – automatically initializes to zero
Done for now
<table>
<thead>
<tr>
<th>Name (Line)</th>
<th>Storage Class</th>
<th>Scope</th>
<th>Linkage</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>w (1)</td>
<td>static</td>
<td>1–8, 17–31</td>
<td>internal</td>
<td>5</td>
</tr>
<tr>
<td>x (2)</td>
<td>static</td>
<td>2–18, 23–31</td>
<td>external</td>
<td>Note a</td>
</tr>
<tr>
<td>func1 (4)</td>
<td>–</td>
<td>4–31</td>
<td>external</td>
<td>–</td>
</tr>
<tr>
<td>a (4)</td>
<td>auto</td>
<td>5–18, 23</td>
<td>none</td>
<td>Note b</td>
</tr>
<tr>
<td>b, c (4)</td>
<td>auto</td>
<td>5–11, 16–23</td>
<td>none</td>
<td>Note b</td>
</tr>
<tr>
<td>d (6)</td>
<td>auto</td>
<td>6–8, 17, 23</td>
<td>none</td>
<td>garbage</td>
</tr>
<tr>
<td>e (6)</td>
<td>auto</td>
<td>6–8, 17–23</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>d (9)</td>
<td>auto</td>
<td>9–11, 16</td>
<td>none</td>
<td>garbage</td>
</tr>
<tr>
<td>e, w (9)</td>
<td>auto</td>
<td>9–16</td>
<td>none</td>
<td>garbage</td>
</tr>
<tr>
<td>b, c, d (12)</td>
<td>auto</td>
<td>12–15</td>
<td>none</td>
<td>garbage</td>
</tr>
<tr>
<td>y (13)</td>
<td>static</td>
<td>13–15</td>
<td>none</td>
<td>2</td>
</tr>
<tr>
<td>a, d, x (19)</td>
<td>register</td>
<td>19–22</td>
<td>none</td>
<td>garbage</td>
</tr>
<tr>
<td>y (20)</td>
<td>static</td>
<td>20–22</td>
<td>external</td>
<td>Note a</td>
</tr>
<tr>
<td>y (24)</td>
<td>static</td>
<td>24–31</td>
<td>internal</td>
<td>zero</td>
</tr>
<tr>
<td>func2 (26)</td>
<td>–</td>
<td>26–31</td>
<td>external</td>
<td>–</td>
</tr>
<tr>
<td>a (26)</td>
<td>auto</td>
<td>27–31</td>
<td>none</td>
<td>Note b</td>
</tr>
<tr>
<td>y (28)</td>
<td>static</td>
<td>28–31</td>
<td>Note c</td>
<td>see y (24)</td>
</tr>
<tr>
<td>z (29)</td>
<td>static</td>
<td>29–31</td>
<td>none</td>
<td>zero</td>
</tr>
</tbody>
</table>

**Note a:** If the variable is not initialized in any other declaration, it will have an initial value of zero.

**Note b:** The initial value of a function parameter is the argument that was passed when the function was called.

**Note c:** The extern keyword doesn’t change the linkage of y that was declared in line 24.