

Autumn 2012

CSE2421 Systems1

Introduction to Low-Level Programming and Computer Organization

Kitty Reeves

TWRF 12:40-1:35pm

TWRF 3:00-3:55pm

CSE2421 = Alice in Wonderland



A look into the rabbit hole...

<http://code.google.com/p/corkami/wiki/PE101?show=content>


Introduction

Website

 <http://www.cse.ohio-state.edu/~reeves>

Syllabus

 Course Description

 Pre-requisites

 Objectives


 Textbook – Safari online an option

 Grading Policy

 Lab locations – submission info given on Day 2

 Academic Misconduct

Thursday *possible* lab day

 BE0310 definitely for the 1st two weeks

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 *****

FYI: Comparing Languages: <http://www.cprogramming.com/langs.html>

If you dare: [http://en.wikipedia.org/wiki/C_\(programming_language\)](http://en.wikipedia.org/wiki/C_(programming_language))

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 hate garbage
 - 🖱 No 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

Welcome to C

- ☰ 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 (whaaaaaaaaaaaaaaaaattt??). 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

```
#include <stdio.h>
void main(void)
{
    printf("Hello, world!\n");
}
```

```
#include <stdio.h>
int main(void) {
    printf("Hello, world!\n");
    return (0); }
```

```
#include <stdio.h>
void main(void) {
    printf("Hello, ");
    printf("world!");
    printf("\n"); }
```

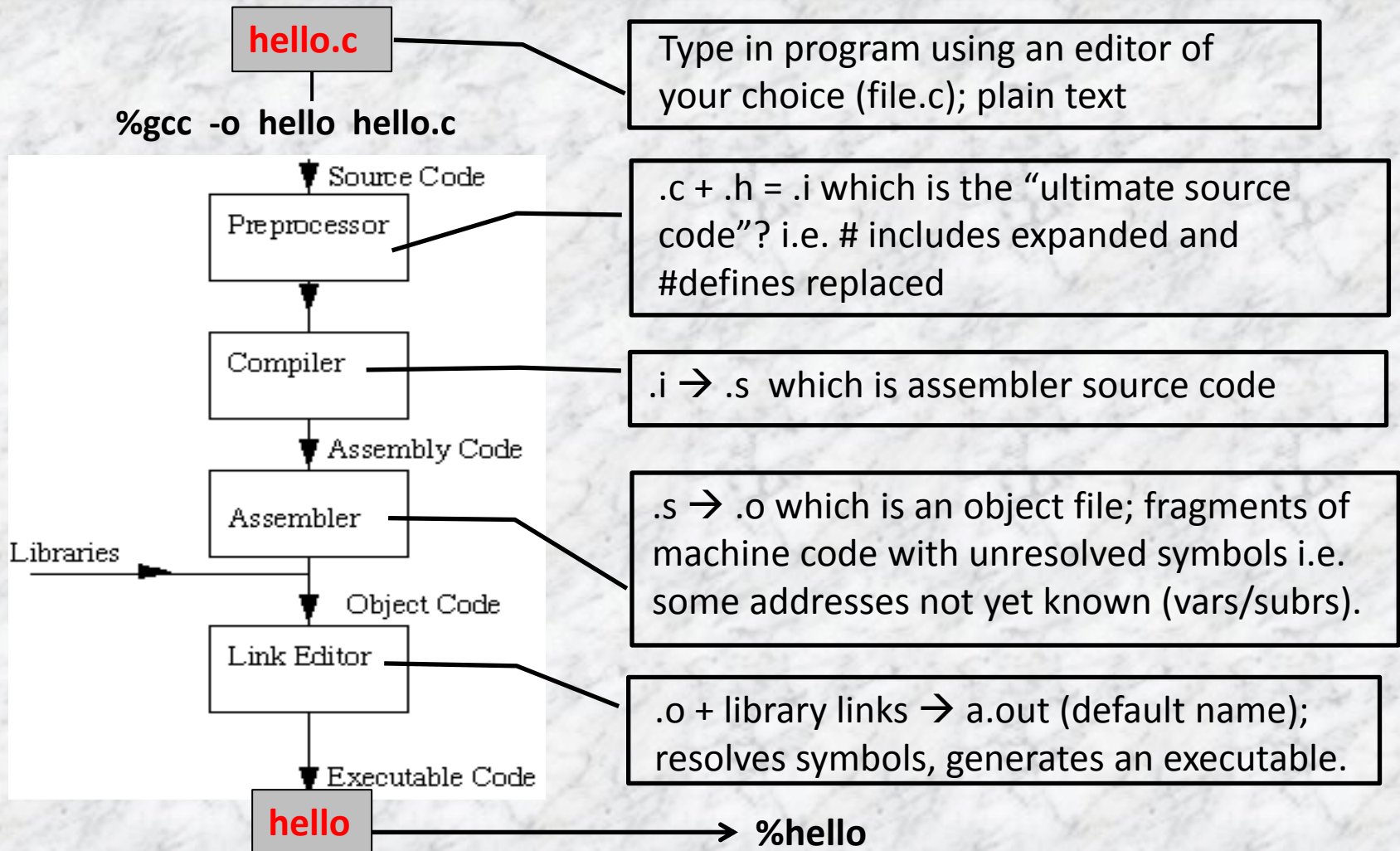
Which one is best?

```
#include <stdio.h>
int main(void) {
    printf("Hello, world!\n");
    getchar();
    return 0; }
```

```
#include <stdio.h>
main() {
    printf("Hello, world!\n");
    return 0; }
```

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



Standard Header Files

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

`<assert.h>` `<float.h>` `<math.h>` `<stdarg.h>` `<stdlib.h>`
`<ctype.h>` `<limits.h>` `<setjmp.h>` `<stddef.h>` `<string.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

The main() function

- Every full C program begins inside a function called "main". A function is simply a collection of commands that do "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.

J

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.

J

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...

Your first C program (cont)

- 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

Your first C program...

New and Improved?

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

/* Main Function
 * Purpose: Controls program, prints Hello, World!
 * Input: None
 * Output: Returns Exit Status
 */
int main(int argc, char **argv) {
    printf("Hello, world!\n");
    return EXIT_SUCCESS;
}
```

- 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.

Global structure

[edit]

After preprocessing, at the highest level a [C program](#) consists of a sequence of declarations at file scope. These may be partitioned into several separate source files, which may be compiled separately; the resulting object modules are then [linked](#) along with implementation-provided run-time support modules to produce an executable image.

The declarations introduce [functions](#), [variables](#) and [types](#). C functions are akin to the subroutines of [Fortran](#) or the procedures of [Pascal](#).

A *definition* is a special type of declaration. A variable definition sets aside storage and possibly initializes it, a function definition provides its body.

An implementation of C providing all of the standard library functions is called a *hosted implementation*. Programs written for hosted implementations are required to define a special function called [main](#), which is the first function called when execution of the program begins.

Hosted implementations start program execution by invoking the `main` function, which must be defined following one of these prototypes:

```
int main() {...}
int main(void) {...}
int main(int argc, char *argv[]) {...}
int main(int argc, char **argv) {...}
```

The first two definitions are equivalent (and both are compatible with C++). It is probably up to individual preference which one is used (the current C standard contains two examples of `main()` and two of `main(void)`, but the draft C++ standard uses `main()`). The return value of `main` (which should be `int`) serves as *termination status* returned to the host environment.

The C standard defines return values 0 and `EXIT_SUCCESS` as indicating success and `EXIT_FAILURE` as indicating failure. (`EXIT_SUCCESS` and `EXIT_FAILURE` are defined in [<stdlib.h>](#)). Other return values have implementation-defined meanings; for example, under [Linux](#) a program killed by a [signal](#) yields a return code of the numerical value of the signal plus 128.

A minimal, correct C program could consist of an empty `main` routine, taking no arguments and returning a constant:

```
int main(void)
{
    return 0;
}
```

The `main` function will usually call other functions to help it perform its job.

Overview of C

- Basic Data Types
- Constants
- Variables
- Identifiers
- Keywords
- Basic I/O

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’

char	1 bytes	-128 to 127
unsigned char	1 bytes	0 to 255
short	2 bytes	-32768 to 32767
unsigned short	2 bytes	0 to 65535
int	4 bytes	-2147483648 to 2147483647
unsigned int	4 bytes	0 to 4294967295
long	4 bytes	-2147483648 to 2147483647
unsigned long	4 bytes	0 to 4294967295
float	4 bytes	1.175494e-38 to 3.402823e+38
double	8 bytes	2.225074e-308 to 1.797693e+308

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)

C Language Variable Types

C Language Variable Types

Whether you're working with regular or unsigned variables in your C program, you need to know a bit about those various variables. The following table show C variable types, their value ranges, and a few helpful comments:

<i>Type</i>	<i>Value Range</i>	<i>Comments</i>
char	-128 to 127	
unsigned char	0 to 255	
int	-32,768 to 32,767	16-bit
	-2,147,483,648 to 2,147,483,647	32-bit
unsigned int	0 to 65,535	16-bit
	0 to 4,294,967,295	32-bit
short int	-32,768 to 32,767	
unsigned short int	0 to 65,535	
long int	-2,147,483,648 to 2,147,483,647	
unsigned long int	0 to 4,294,967,295	
float	1.17×10^{-38} to 3.40×10^{38}	6-digit precision
double	2.22×10^{-308} to 1.79×10^{308}	15-digit precision

Derived types

- 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

```
char          'A', 'B'
int           123, -1, 2147483647, 040 (octal), 0xab (hexadecimal)
unsigned int  123u, 2107433648, 040U (octal), 0X02 (hexadecimal)
long         123L, 0x1FFF1 (hexadecimal)
unsigned long 123ul, 0777UL (octal)
float        1.23F, 3.14e+0f
double       1.23, 2.718281828
long double  1.23L, 9.99E-9L
```



Special characters

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

<code>\a</code>	alert (bell) character	<code>\\</code>	backslash
<code>\b</code>	backspace	<code>\?</code>	question mark
<code>\f</code>	formfeed	<code>\'</code>	single quote
<code>\n</code>	newline	<code>\"</code>	double quote
<code>\r</code>	carriage return	<code>\ooo</code>	octal number
<code>\t</code>	horizontal tab	<code>\xhh</code>	hexadecimal number
<code>\v</code>	vertical tab		

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:
 - 🕒 The use of `EXIT_SUCCESS` in `hello.c` code
 - 🕒 `#define PI 3.141593`
 - 🕒 `#define TRUE 1`
 - 🕒 `#define floatingpointnum float`

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? */

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

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

- 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.

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

```
#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 */
```

```
#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.

C Language Conversion Characters

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:

Conversion Character	Displays Argument (Variable's Contents) As
%c	Single character
%d	Signed decimal integer (int)
%e	Signed floating-point value in E notation
%f	Signed floating-point value (float)
%g	Signed value in %e or %f format, whichever is shorter
%i	Signed decimal integer (int)
%o	Unsigned octal (base 8) integer (int)
%s	String of text
%u	Unsigned decimal integer (int)
%x	Unsigned hexadecimal (base 16) integer (int)
%%	(percent character)

Answers: check.c and check1.c

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

```
#include <stdio.h>
#define PI 3.14159265358979323846
int main() {
    int x;
    scanf("%d", &x); /* why need & ? */
    printf("%d\n", x);
    float var;
    scanf("%f",&var);
    scanf("%d",&var);
    scanf("%lf", &var);
    int first, second;
    scanf("enter value ", &var);
    scanf("%d%d", &first, &second);
    int i, j;
    scanf(" %d %*d %*d%*d %d ", &i, &j)
    return 0; }
```


Printf formatted output conversions

Character	Argument type; Printed As
d,i	int; signed decimal notation.
o	int; unsigned octal notation (without a leading zero).
x,X	unsigned int; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef for 0x or ABCDEF for 0X.
u	int; unsigned decimal notation.
c	int; single character, after conversion to unsigned char
s	characters from the string are printed until a '\0' is reached or until the number of characters indicated by the precision have been printed.
f	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.
e,E	double; decimal notation of the form [-]m.dddddde+/-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.
g,G	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.

Decimal & Floating point

- Width of the whole number portion
 - Is for decimal integers
- The character width for float
 - includes the decimal point position

%d	print as decimal integer
%6d	print as decimal integer, at least 6 characters wide
%f	print as floating point
%6f	print as floating point, at least 6 characters wide
%.2f	print as floating point, 2 characters after decimal point
%6.2f	print as floating point, at least 6 wide and 2 after decimal point

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\nPI=%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 (cont)

- 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

FORMAT	MEANING	VARIABLE TYPE
%d	read an integer value	int
%ld	read a long integer value	long
%f	read a real value	float
%lf	read a double precision real value	double
%c	read a character	char
%s	read a character string from the input	array of char

Scanf examples

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

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

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

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

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

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

```
string s;  
scanf("%9s", s);  
Input:  
VeryLongString  
s=="VeryLongS"
```

NOTE: pressing the enter key means you have entered a character...

more Scanf examples

Letter	Type of Matching Argument	Auto-skip; Leading White-Space	Example	Sample Matching Input
%	% (a literal, matched but not converted or assigned)	no	int anInt; scanf("%i%%", &anInt);	23%
d	int	yes	int anInt; long l; scanf("%d %ld", &anInt, &l);	-23 200
i	int	yes	int anInt; scanf("%i", &anInt);	0x23
o	unsigned int	yes	unsigned int aUInt; scanf("%o", &aUInt);	023
u	unsigned int	yes	unsigned int aUInt; scanf("%u", &aUInt);	23
x	unsigned int	yes	unsigned int aUInt; scanf("%x", &aUInt);	1A
a, e, f, g	float or double	yes	float f; double d; scanf("%f %lf", &f, &d);	1.2 3.4
c	char	no	char ch; scanf(" %c", &ch);	Q
s	array of char	yes	char s[30]; scanf("%29s", s);	hello
n	int	no	int x, cnt; scanf("X: %d%n", &x, &cnt);	X: 123 (cnt==6)
[array of char	no	char s1[64], s2[64]; scanf(" %[^\\n]", s1); scanf("%[^\\t] %[^\\t]", s1, s2);	Hello World field1 field2

Scanf (cont)

■ You can use this function even with spaces in the input:

👁 `scanf(" %[^\n]s",a);`

C Language operators

C Language Operators

In programming with C, you occasionally want to use common mathematical operators for common mathematical functions and not-so-common operators for logic and sequence functions. Here's a look at C language operators to use:

Operator, Category, Duty	Operator, Category, Duty	Operator, Category, Duty
=, Assignment, Equals	!=, Comparison, Is not equal to	>, Bitwise, Shift bits right to
+, Mathematical, Addition	&&, Logical, AND	~, Bitwise, One's complement
-, Mathematical, Subtraction	, Logical, OR	+, Unary, Positive
*, Mathematical, Multiplication	!, Logical, NOT	-, Unary, Negative
/, Mathematical, Division	++, Mathematical, Increment by 1	*, Unary, Pointer
%, Mathematical, Modulo	--, Mathematical, Decrement by 1	&, Unary, Address
>, Comparison, Greater than	&, Bitwise, AND	sizeof, Unary, Returns the size of an object
>=, Comparison, Greater than or equal to	, Bitwise, Inclusive OR	., Structure, Element access
<, Comparison, Less than	^, Bitwise, Exclusive OR (XOR or EOR)	->, Structure, Pointer element access
<=, Comparison, Less than or equal to	<<, Bitwise, Shift bits left	?:, Conditional, Funky if operator expression
==, Comparison, Is equal to		

Arithmetic type issues

■ Type combination and promotion

🖱 ('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

Arithmetic operators

■ 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

👁 * / %

👁 + -

Arithmetic type conversions

- 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.

Arithmetic Expressions – A bug's life

Pitfall -- int Overflow

"I once had a piece of code which tried to compute the number of bytes in a buffer with the expression $(k * 1024)$ where k was an int representing the number of kilobytes I wanted. Unfortunately this was on a machine where int happened to be 16 bits. Since k and 1024 were both int, there was no promotion. For values of $k \geq 32$, the product was too big to fit in the 16 bit int resulting in an overflow. The compiler can do whatever it wants in overflow situations -- typically the high order bits just vanish. One way to fix the code was to rewrite it as $(k * 1024L)$ -- the long constant forced the promotion of the int. This was not a fun bug to track down -- the expression sure looked reasonable in the source code. Only stepping past the key line in the debugger showed the overflow problem. "Professional Programmer's Language." This example also demonstrates the way that C only promotes based on the **types** in an expression. The compiler does not consider the values 32 or 1024 to realize that the operation will overflow (in general, the values don't exist until run time anyway). The compiler just looks at the compile time types, int and int in this case, and thinks everything is fine."

Arithmetic expressions - Truncation

Pitfall -- int vs. float Arithmetic

Here's an example of the sort of code where int vs. float arithmetic can cause problems. Suppose the following code is supposed to scale a homework score in the range 0..20 to be in the range 0..100.

```
{
int score;
...// suppose score gets set in the range 0..20 somehow
7
score = (score / 20) * 100; // NO -- score/20 truncates to 0
...
```

Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20. The fix is to force the quotient to be computed as a floating point number...

```
score = ((double)score / 20) * 100; // OK -- floating point division from cast
score = (score / 20.0) * 100; // OK -- floating point division from 20.0
score = (int)(score / 20.0) * 100; // NO -- the (int) truncates the floating
// quotient back to 0
```

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;
}
```

-  Variables
-  Function calls
-  Input
-  Output
-  Operators
-  Typcasting

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);`

Example

```
#include <stdio.h>
/* print Fahrenheit-Celsius table for fahr = 0, 20, ..., 300
   where the conversion factor is  $C = (5/9) \times (F-32)$  */
main()
{
    int fahr, celsius;
    int lower, upper, step;
    lower = 0;           /* lower limit of temperature scale */
    upper = 300;        /* upper limit */
    step = 20;          /* step size */
    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;    // problem? 9.0? Typecast?
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step; }
    return 0;
}
```

Example

```
#include <stdio.h>
#define MAGIC 10
int main(void)
{
    int i, fact, quotient;
    while (i++ < 3)           // value of i? need to initialize
    {
        printf("Guess a factor of MAGIC larger than 1: ");
        scanf("%d", &fact);
        quotient = MAGIC % fact;
        if (0 == quotient)
            printf("You got it!\n");
        else
            printf("Sorry, You missed it!\n");
    }
    return 0;
}
```

i++ is the same as:

i = i + 1

How evaluate?

i = i + 1 < 3

3 1 2

Problem, but...

(i = i + 1) < 3

Assignment operator

- 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
- Example1: `a=x=y+3;` so `a = x`, right?
- Example2: `r=s+(t=u-v)/3;` give “same as” code.

Same as:
`c=1;`
`b=c;`
`a=b;`

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.

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

Logical/Boolean Operators (cont)

- 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

Operator	Operator's Name	Example	Result
&&	AND	3>2 && 3>1	1(true)
&&	AND	3>2 && 3<1	0(false)
&&	AND	3<2 && 3<1	0(false)
	OR	3>2 3>1	1(true)
	OR	3>2 3<1	1(true)
	OR	3<2 3<1	0(false)
!	NOT	!(3==2)	1(true)
!	NOT	!(3==3)	0(false)

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)

Loop constructs

```
for (init; cond; modify) {  
    statement(s);  
}
```

```
while (cond) {  
    statement(s);  
}
```

```
do {  
    statement(s);  
} while (cond);
```

- 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.

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

```
int main () {  
    int n, sum;  
    sum = 0;  
    for (n = 1; n <= 10; n=n+1)  
        {  
            sum = sum + n;  
        }  
    printf("\n The sum of integers from 1 to 10 is %d, have a nice day", sum);  
    return 0;  
}
```

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

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

Loop construct examples

```
#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 */

```
#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; }
```

```
#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*/
```


Break, Continue, goto

- 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.

Break, Continue EXAMPLES

```
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); }    /* 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); }    /* 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

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

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

```
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

```
#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;
}
```

/ Need a variable... */*
/ Asks for age */*
/ The input is put in age */*
/ If the age is less than 100 */*
/ Just to show you it works... */*
/ use else to show an example */*
/ how rude! */*
/ do this if no other block exec */*

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).

Switch example

```
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; }
```

```
#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; }
```

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)

Using GDB

- `%nl gdbincl.c > gdbinclnl`
 - 🔗 `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`
- COMMANDS/tutorial
 - <http://www.yolinux.com/TUTORIALS/GDB-Commands.html>
 - <http://www.cprogramming.com/gdb.html>
 - 🔗 `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

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) ...

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

```
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.

```
/* 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.

Unary Operators: ++ --

- ++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...

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

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


Unary Operators ++ and -- (cont)

```
int i = 1, j = 1;  
printf("\t%d %d\n", ++i, j++);  
printf("\t%d %d\n", i, j);
```

Output:

```
2 1  
2 2
```

```
i=1; j=1;  
printf("\t%d \n", i=j++);  
printf("\t%d \n", i=++j);
```

Output:

```
1  
3
```

```
i = 0; j = 0;
```

```
if ( (i++ == 1) && (j++ == 1))  
    printf("what will happen?\n");  
printf("\t%d %d\n", i, j);
```

Will i and j get incremented?

2nd printf output: 1 0

The answer is NO! Because the expression in the left of '&&' resolves to false the compiler does NOT execute the expression on the right and so 'j' does not get executed!

Size of operator

- **sizeof** will return the number of bytes reserved for a variable or data type.
- Returning the length of a data type (ex.1)
- Length of a variable (ex.2)
- Number of bytes reserved for a structure (ex.3)
- `size_t` is type designation for the unsigned integer result of the `sizeof` keyword

Example 1:

```
/* How big is an int?  
expect an answer of 4. */  
main() {  
    printf("%d \n", sizeof(int));
```

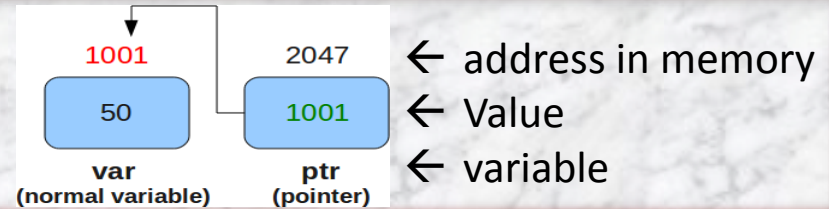
Example 2:

```
main() {  
    char String[20];  
    printf ("%d \n", sizeof String);  
    printf ("%d \n", sizeof (String)); }  
// brackets optional but rec
```

Example 3:

```
/* Will print 8 on most machines. */  
main()  
{  
    struct  
    {  
        int a;  
        int b;  
    } TwoInts;  
  
    printf("%d \n", sizeof(TwoInts));  
}
```

Pointer definition



- Values of variables are stored in memory, at a particular location
- A location is identified and referenced with an address
 - 🔗 Analogous to identifying a house's location via an address
- A pointer is a variable that contains the address of another variable
- * is used in the declaration of a pointer type
 - 🔗 **int *p** means variable p is a pointer that points to an integer
- & (unary operator) gives the "address of" an object
 - 🔗 **p = &c** means the address of c is assigned to the variable p
- * (unary not arithmetic operator) is a dereferencing operator when applied to pointers
 - 🔗 When applied to a pointer, it accesses the object the pointer points to
 - 🔗 * in front of a pointer variable means "get the value at that address" i.e. "contents of"
 - 🔗 **int a = *p** means get the value at the address designated by p and assign it to
 - 🔗 ***p = 1** means assign the value of 1 to the memory location designated by the address of p
- Every pointer points to a specific data type
 - 🔗 Exception = void (a generic pointer); pointer to void holds any type of pointer but can't be dereferenced (i.e. cannot get the "contents of")

J

Ah, yes. POINTERS. At last, we arrive at THE MOST DREADED WORD in the lexicon of the C student. Pointers are indeed so dreaded that Java has completely done away with pointers and wrapped their functionality into the (admittedly safer) concept of *references*. C++, as a transitional step, has both pointers and references.

Declaring Pointers

int* ptr_a;

int *ptr_a;

The first style leads to mistakes

int* ptr_b, ptr_c, ptr_d

➤ b is a pointer but c and d are integers

int *ptr_b, *ptr_c, *ptr_d

➤ 3 pointers are declared here

Char example

char ch = 'c';

char *chptr = &ch;

char *ptr = chptr;

➤ see last example in previous slide

Pointer example

Reminders:

- * in a declaration says "I am a pointer" that points to a certain type of value
- & "address of"
- * In front of a pointer type says "get the value at that address" i.e. "contents of" operator

EXAMPLE:

```
int x=1, y=2, z[10];  
int *ip;  
ip = &x;  
y = *ip;  
*ip = 0;  
ip = &z[0];
```

VARIABLE	ADDRESS (in decimal)	MEMORY (assuming 4 bytes per word and each block is a byte)*
ip	0	Is a pointer; holds an addr; 8... 16
	4	
x	8	1... 0
y	12	2... 1
z	16	z[0]
	20	z[1]
	24	z[2]
	28	etc
	32	
	36	
	40	
	44	

* not going to worry about "size" right now

Pointer examples... more!

- Every pointer points to a specific data type.
 - 🕒 one exception:
 - 🕒 a “pointer to void” is used to hold any type of pointer but cannot be dereferenced itself (later)

- If ip points to the integer x ($ip = \&x$) then $*ip$ can occur in any context where x could
 - 🕒 Example: $*ip = *ip + 10 \rightarrow x = x + 10$; increments the contents of the address at ip by 10

- The unary operators $*$ and $\&$ bind more tightly than arithmetic operators
 - 🕒 Example: $y = *ip + 1$ takes whatever ip points at, adds 1, and assigns the result to y
 - 🕒 Other ways to increment by 1:
 - $*ip += 1 \rightarrow *ip = *ip + 1$
 - $++*ip$
 - $(*ip)++$
 - The parentheses are necessary; without them, the expression would increment ip instead of what it points to, because unary operators like $*$ and $++$ associate right to left.

- Pointers are variables so can be used without dereferencing.
 - 🕒 Example:

```
int *iq, *ip
iq = ip
```

 - copies the contents of ip (an address) into iq, thus making iq point to whatever ip pointed to.

You try...

```
/* EXAMPLE 3 */
#include <stdio.h>
int main(void) {
    char ch = 'c';
    char *chptr = &ch;
    int i = 20;
    int *intptr = &i;
    float f = 1.20000;
    float *fptr = &f;
    char *ptr = "I am a string";
    printf("\n [%c], [%d], [%f], [%c], [%s]\n", *chptr, *intptr, *fptr, *ptr, ptr);
    return 0; }
```

```
/* EXAMPLE 1 */
#include<stdio.h>
int main() {
    float i=10, *j;
    void *k;
    k=&i;
    j=k;
    printf("%f\n", *j);
    return 0; }
```

```
/* EXAMPLE 2 */
#include <stdio.h>
#include <stdlib.h>
main() {
    int x, *p;
    p = &x;
    *p = 0;
    printf("x is %d\n", x);
    printf("*p is %d\n", *p);
    *p += 1;
    printf("x is %d\n", x);
    (*p)++;
    printf("x is %d\n", x);
    return 0; }
```

C functions

- Similar to Java methods but...
 - 🖱 Not part of a class
 - 🖱 Not associated with an object
 - 🖱 No “this”

Function definition

■ Function prototype

👁 Return type*

👁 Argument definition

👁 return_type function_name (type1 arg1,type2 arg2,...,typen argn)

```
int add(int p,int q);
```

■ Function calls

👁 Basic syntax

👁 Parameter passing*

```
z = add(a,b);
```

■ Standard library and function calls

NOTES:

- Functions should be short and sweet.
- Functions do not nest.
- Variables have to be communicated through function arguments or global variables.
- * If no return type or no argument type, then it defaults to int

```
int add(int p,int q)  
{  
    return p+q;  
}
```

Sample: <http://www.cprogrammingexpert.com/images/Function.gif>

Pointers and Functions

■ Pass By Value

- 👁️ Passing a variable by value makes a copy of the variable before passing it onto a function. This means that **if you try to modify the value inside a function, it will only have the modified value inside that function.** Once the function returns, the variable you passed it will have the same value it had before you passed it into the function.

■ Pass By Reference

- 👁️ There are two instances where a variable is passed by reference:
 - When you modify the value of the passed variable locally (inside the callee) and the value of the variable in the calling function as well.
 - To avoid making a copy of the variable for efficiency reasons.
- 👁️ Technically, C does not “pass by reference” as typically seen in other programming languages. Actually, when you pass a pointer (an address) , a copy is made of that variable so two variables point to the same address (one from the callee and one from the caller).

Function Prototypes

- A number of statements grouped into a single logical unit are called a function
- REMINDER → It is necessary to have a single function 'main' in every C program.
- A function prototype is a function declaration or definition which includes:
 - 🕒 Information about the number of arguments
 - 🕒 Information about the types of the arguments
- Although you are allowed not to specify any information about a function's arguments in a declaration, it is purely because of backwards compatibility with Old C and should be avoided (poor coding style).
 - 🕒 A declaration without any information about the arguments is *not* a prototype.



Only one function with a given name may be defined. Unlike Java, C does not support overloading (i.e., two functions with the same name but different signatures).

Pointers and Function arguments

Call: swap(a,b);

```
/* WRONG */  
void swap(int x, int y)  
{  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}
```

Call: swap(&a,&b);

```
/* interchange *px and *py */  
void swap(int *px, int *py)  
{  
    int temp;  
    temp = *px;  
    *px = *py;  
    *py = temp;  
}
```

- Since C passes arguments to functions by value and make a copy local to swap; so there is no direct way for the called function (callee) to alter a variable in the calling function (caller).
- Because of call by value, swap can't affect the arguments a and b in the routine that called it.
- The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:
- Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.
- NOW KNOW WHY SCANF NEEDS & SYMBOLS!!!

The RETURN statement

- Every function except those returning void should have at least one, each return showing what value is supposed to be returned at that point.
- Although it is possible to return from a function by falling through the last }, unless the function returns void, an unknown value will be returned, resulting in undefined behavior.
- The type of expression returned must match the type of the function, or be capable of being converted to it as if an assignment statement were in use.
- Following the return keyword with an expression is *not* permitted if the function returns void.

Another Function example

```
#include <stdio.h>
#include <stdlib.h>
main ()          /* was-could be void pmax(); now considered bad practice */
{ void pmax(int first, int second);          /* declaration prototype */
  int i,j;
  for(i = -10; i <= 10; i++)
  {   for(j = -10; j <= 10; j++)
      {       pmax(i,j);
          }
      }
  return 0;
}
/* Prints larger of its two arguments. */
void pmax (int a1, int a2)          /* definition */
{   int biggest;
    if (a1 > a2)
        { biggest = a1; }
    else{ biggest = a2; }
    printf("larger of %d and %d is %d\n", a1, a2, biggest);
}
```

Function example

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

void printtotal(int total);
void addxy(int x, int y, int total);
void subxy(int x, int y, int *total);

void main() {
    int x, y, total;
    x = 10;
    y = 5;
    total = 0;
    printtotal(total);
    addxy(x, y, total);
    printtotal(total);
    subxy(x, y, &total);
    printtotal(total); }
```

Program continued...

```
void printtotal(int total) {
    printf("Total in Main: %dn", total);
}

void addxy(int x, int y, int total) {
    total = x + y;
    printf("Total from inside addxy: %dn",
total); }

void subxy(int x, int y, int *total) {
    *total = x - y;
    printf("Total from inside subxy: %dn",
*total);
}
```

Another Function example

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

void date(int *, int *); /* declare the function */

main() {
    int month, day;
    date (&day, &month);
    printf("day is %d, month is %d\n", day, month);
    return 0;}

void date(int *day_p, int *month_p) {
    int day_ret, month_ret;
    /* * At this point, calculate the day and month *
       values in day_ret and month_ret respectively. */
    *day_p = day_ret;
    *month_p = month_ret; }
```


Function Summary

- Functions can be called recursively.
- Functions can return any type that you can declare, except for arrays and functions (you can get around that restriction to some extent by using pointers).
- Functions returning no value should return void.
- Always use function prototypes.
- Undefined behavior results if you call or define a function anywhere in a program unless either
 - 👁 a prototype is *always* in scope for *every call* or definition, or
 - 👁 you are very, very careful.
- Assuming that you *are* using prototypes, the values of the arguments to a function call are converted to the types of the formal parameters exactly as if they had been assigned using the = operator.
- Functions taking no arguments should have a prototype with (void) as the argument specification.

Declaration of Arrays

- An array is a way to store many values under the same name in adjacent memory locations.
- Arrays must be declared before they can be used in the program.
- Standard array declaration is as
 - 👁 `<type> <name> [<size>];`
 - 👁 `<size>` elements i.e. values of the array, are stored using an index/subscript number from 0 to `<size>-1`
- Examples
 - 👁 `double height[10];` // height[0] to height[9]
 - 👁 `float width[20];` //width[0] to width[19]
 - 👁 `int min[9];` // etc
 - 👁 `char name[20];` // a string!
- **Why first index/subscript=0???**
 - 👁 Address of min = address of min[0]

in memory:												
min -->	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]			
address -->	+0	+4	+8	+12	etc							

Index checking

- **Index access is not checked by the compiler**
 - 🕒 **Check for valid range manually**
 - 🕒 **Especially important for user entered indices**
- Index checking means that, in all expressions indexing an array, first check the index value against the bounds of the array which were established when the array was defined, and should an index be out of bounds, further execution is suspended via some sort of error (buffer overflow, segmentation fault, bug).
- Important to understand how arrays are used “behind the scenes”
- Performing bounds checking during every usage is time-consuming
- C never performs automatic bounds checking in order to raise speed
- It depends on the OS to ensure that you are accessing valid memory.
- There’s a difference in being outside array bounds but inside your allotted memory; and outside the array bounds and outside your allotted memory!
- Yet... sizeof (array) works, but that’s the total number of bytes not the index bounds themselves

Initializing Arrays

- The initializing values are enclosed within the curly braces in the declaration and placed following an equal sign after the array name.
- Initialize an individual array location (name[sub]) like any other variable/memory location.
- An array location can be used like any other single variable:
 - 👁 `x = array[3]`
 - 👁 `array[5]=x+y`

```
int studentAge[4];
studentAge[0]=14;
studentAge[1]=13;
studentAge[2]=15;
studentAge[3]=16;
```

```
//initialize and print all the elements of the array
int myArray [5] = {1,2,3,4,5};
for (int i=0;i<5;i++)
{   printf("%d", myArray[i]);
}
```


Copying Arrays

- There is no such statement in C language which can directly copy an array into another array. So we have to copy each item separately into another array.

```
#include <stdio.h>
int main()
{   int iMarks[4] = {78, 64, 66, 74};
    int newMarks[4];
    int i,j;
    for(i=0; i<4; i++)
        newMarks[i]=iMarks[i];
    for(j=0; j<4; j++)
        printf("%d\n", newMarks[j]);
    return 0; }
```

Manipulating Arrays

- C Language treats the name of the array as if it were a pointer to the first element
 - 📎 see handout ArrayInOutSwapReverse.docx
- The name of the array refers to the whole array. It works by representing a pointer to the start of the array.

Prototype/Call

```
void intSwap(int *x, int *y)
intSwap(&a[i], &a[n-i-1]);
```

```
void printIntArray(int a[], int n)
printIntArray(x, hmny);
```

```
int getIntArray(int a[], int nmax, int sentinel)
hmny = getIntArray(x, 10, 0);
```

```
void reverseIntArray(int a[], int n)
reverseIntArray(x, hmny);
```

When we pass arrays into functions, the compiler automatically converts the array into a pointer to the first element of the array. In short, **the array without any brackets will act like a pointer**. So we just pass the array directly without using the ampersand.

Multi-dimensional Arrays

- Declarations – [row][col] subscript order
 - 🖱 float table [50] [50];
 - 🖱 char line [24] [40];
 - 🖱 int values [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }
- **How stored?** ➔ **row order**

		columns			
		0	1	2	3
rows	0	1	2	3	4
	1	5	6	7	8
	2	9	10	11	12

in memory:											
values -->	[0][0]	[0][1]	[0][2]	[0][3]	[1][0]	[1][1]	[1][2]	[1][3]	[2][0]	[2][1]	etc
address -->	+0	+4	+8	+12	etc						

Multi-dimensional Arrays Example

```
#include <stdio.h>
int main()
{   int x; int y; int array[8][8];   /* Declares an array like a gameboard or matrix*/
    for ( x = 0; x < 8; x++ )
    {   for ( y = 0; y < 8; y++ )
        array[x][y] = x * y;        /* Set each element to a value */
    }
    printf( "Array Indices:\n" );
    for ( x = 0; x < 8;x++ )
    {   for ( y = 0; y < 8; y++ )
        {   printf( "[%d][%d]=%d", x, y, array[x][y] );
            }
        printf( "\n" );
    }
    getchar();
    return 0;
}
```


Character Arrays i.e. Strings

■ Declarations:

🖱️ `char arr[] = {'c','o','d','e','\0'};`

- The null byte is required as a terminating byte when string is read as a whole.

🖱️ `char arr[] = "code";`

- Implies that there are 4 characters along with the NUL byte (i.e. the `\0` character) so a “length” of 5.

■ This type of array allocation, where the size of the array is determined at compile-time, is called *static allocation*.

Pointers and Strings

- A string is an array of characters.
 - 🕒 So we have no string pointers in C. Its the character pointers that are used in case of strings too.
 - 🕒 When we point a pointer to a string, by default it holds the address of the first character of the string (just like an array)
- Gives the memory address without a reference operator(&)
 - 🕒 `char *ptr;`
 - 🕒 `char str[40];`
 - 🕒 `ptr = str;`
- Strings end with an implied `\0` by default
 - 🕒 `"I am a string" = I_am_a_string\0`
 - 🕒 `sizeof` operator says size = ??
 - 🕒 `strlen()` function is in the `string.h` header file
 - 🕒 The `strlen` function returns the length of the null-terminated string `s` in bytes. In other words, it returns the **offset (i.e. starting at position zero)** of the terminating null character within the array.
 - `char string[32] = "hello, world";`
 - `sizeof (string) ⇒ 32`
 - `strlen (string) ⇒ 12`
 - this will not work unless *string* is the character array itself, not a pointer to it

Character Array (i.e. string) example

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

int main(void)
{   char arr[4];           // for accommodating 3 characters and one null '\0' byte
    char *ptr = "abc";    // a string containing 'a', 'b', 'c', '\0'

    //reset all the bytes so that none of the byte contains any junk value
    memset(arr, '\0', sizeof(arr));

    strncpy(arr, ptr, sizeof("abc"));    // Copy the string "abc" into the array arr
    printf ("\n %s \n",arr);            // print the array as string
    arr[0] = 'p';                        // change the first character in the array
    printf("\n %s \n",arr);              // again print the array as string
    return 0;
}
```

Dynamic Memory Functions

- Can be found in the `stdlib.h` library:
 - 🖱 To allocate space for an array in memory you use
 - `calloc()`
 - 🖱 To allocate a memory block you use
 - `malloc()`
 - 🖱 To de-allocate previously allocated memory you use
 - `free()`
- Each function is used to initialize a pointer with memory from free store (a section of memory available to all programs)

malloc

- The function malloc() will allocate a block of memory that is size bytes large. If the requested memory can be allocated a pointer is returned to the beginning of the memory block.
 - **Note:** the content of the received block of memory is not initialized.
- **Usage of malloc():**
 - 🔗 void * malloc (size_t size);
- **Parameters:**
 - 🔗 Size of the memory block in bytes.
- **Return value:**
 - 🔗 If the request is successful then a pointer to the memory block is returned.
 - 🔗 If the function failed to allocate the requested block of memory, a null pointer is returned.
- **Example**
 - 🔗 <http://www.codingunit.com/c-reference-stdlib-h-function-malloc>
- **Another example:**
 - 🔗 **#include <stdlib.h>**
 - 🔗 **int *ptr = malloc(sizeof (int));**
 - set ptr to point to a memory address of size int
 - 🔗 **int *ptr = malloc(sizeof (*ptr));**
 - is slightly cleaner to write malloc statements by taking the size of the variable pointed to by using the pointer directly
 - 🔗 **float *ptr = malloc(sizeof (*ptr));**
 - float *ptr;
 - /* hundreds of lines of code */
 - ptr = malloc(sizeof(*ptr));

calloc

■ Usage of calloc():

👁 void * calloc (size_t num, size_t size);

■ Parameters:

👁 Number of elements (array) to allocate and the size of elements.

■ Return value:

👁 Will return a pointer to the memory block. If the request fails, a NULL pointer is returned.

■ Example:



- <http://www.codingunit.com/c-reference-stdlib-h-function-calloc>
- note: ptr_data = (int*) calloc (a,sizeof(int));

Difference Between Malloc and Calloc


- The number of arguments. `malloc()` takes a single argument (memory required in bytes), while `calloc()` needs two arguments.
- `malloc()` does not initialize the memory allocated, while `calloc()` initializes the allocated memory to ZERO.

FYI... the exit function

Syntax:

-  #include <stdlib.h>
-  void exit(int exit_code);

Description:

-  The exit() function stops the program. *exit_code* is passed on to be the return value of the program, where usually zero indicates success and non-zero indicates an error.

Example:

-  <http://www.codingunit.com/c-reference-stdlib-h-function-exit>

Static and Dynamic Arrays

- Static arrays are used when we know the amount of bytes in array at *compile time*.
 - 🕒 Static arrays are ones that reside *on the stack*
 - 🕒 `char arr[10];`
- A dynamic array is used where we come to know about the size on *run time*.
 - 🕒 Dynamic arrays is a popular name given to a series of bytes allocated *on the heap*.
 - 🕒 `char *ptr = (char*) malloc(10);`
 - 🕒 allocates a memory of 10 bytes on heap and we have taken the starting address of this series of bytes in a character pointer ptr.
 - 🕒 Fine if know number of characters, but what if don't?
 - Read in one char/byte at a time until the user presses the enter key
- **malloc** (memory allocation) is used to dynamically allocate memory at run time. Possible uses for this function are:
 - 🕒 Read records of an unknown length.
 - 🕒 Read an unknown number of database records.
 - 🕒 Link lists.

free

- The free function returns memory to the operating system.
- 👁 **free(ptr);**
- 👁 After freeing a pointer, it is a good idea to reset it to point to 0.

NOTE: When 0 is assigned to a pointer, the pointer becomes a null pointer...in other words, it points to nothing. By doing this, when you do something foolish with the pointer (it happens a lot, even with experienced programmers), you find out immediately instead of later, when you have done considerable damage.

Pointer Arithmetic

- When you add to or subtract from a pointer, the amount by which you do that **is multiplied by the size of the type** the pointer points to.
- In the case of our three increments, each 1 that you added was multiplied by sizeof(int).

```
int array[] = { 45, 67, 89 };
int *array_ptr = array;
printf(" first element: %i\n", *(array_ptr++));1
printf("second element: %i\n", *(array_ptr++));
printf(" third element: %i\n", *array_ptr);
```

Output:

```
first element: 45
second element: 67
third element: 89
```

NOTE 1: 1==4 (programmer humor?!)

```
*(array_ptr++) == *array_ptr++
```

**B
T
W**

`*(array_ptr++)`¹

VS

`(*array_ptr)++`

find the value at that address, output, then add "1" to the address

VS

Find the value at the address, output, then add one to the value at that address

Pointer Arithmetic (cont)

Expression	Assuming p is a pointer to a...	... and the size of *p is...	Value added to the pointer
p+1	char	1	1
p+1	short	2	2
p+1	int	4	4
p+1	double	8	8
p+2	char	1	2
p+2	short	2	4
p+2	int	4	8
p+2	double	8	16

Pointer Arithmetic (again)

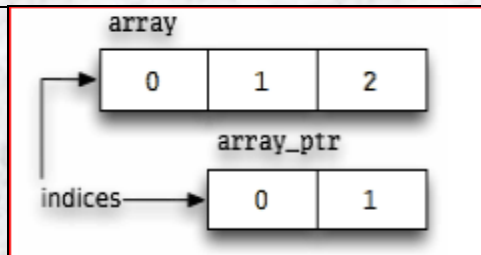
- pointer (+ or -) integer
 - Only for pointers that are pointing at an element of an array
 - Also works with malloc
 - Watch for bounds (begin and end)
 - Ok to go one beyond the array but not a valid dereference
- pointer#1 – pointer#2
 - Only allowed when both point to elements of the same array and $p1 \text{ index} < p2 \text{ index}$
 - Measured in array elements not bytes
 - If $p1 \rightarrow \text{array}[i]$ and $p2 \rightarrow \text{array}[j]$ then $p2 - p1 == j - i$

Pointer Indexing

```
int array[] = { 45, 67, 89 };  
printf("%i\n", array[0]); // output is 45  
// array and array[0] point to same thing
```

- The subscript operator (the [] in array[0]) has *nothing to do with arrays*.
- In most contexts, arrays decay to pointers. This is one of them: That's a *pointer* you passed to that operator, not an array.

```
int array[] = { 45, 67, 89 };  
int *array_ptr = &array[1];  
printf("%i\n", array_ptr[1]);  
//output is 89 (whooooooooooaaahhhhtttt??!!)
```



- array points to the first element of the array;
 - 🕒 `array[1] == *(array + 1)`
- array_ptr is set to `&array[1]`, so it points to the second element of the array.
- So array_ptr[1] is equivalent to array[2]

NULL vs 0 vs '\0'

- NULL is a macro defined in several standard headers
- 0 is an integer constant
- '\0' is a character constant, and
 - 👁 nul is the name of the character constant.

All of these are **not interchangeable**

- NULL is to be used for pointers only since it may be defined as `((void *) 0)`, this would cause problems with anything but pointers.
- 0 can be used anywhere, it is the generic symbol for each type's zero value and the compiler will sort things out.
- '\0' should be used only in a character context.
 - 👁 nul is not defined in C or C++, it shouldn't be used unless you define it yourself in a suitable manner, like:
 - `#define nul '\0'`

NULL pointer and VOID

- 0 (an integer value) is convertible to a null pointer value if assigned to a pointer type
- VOID – no value at all – literally means “nothing”
 - 🔗 So it is type-less (no type defined) so can hold any type of pointer
 - 🔗 We cannot perform arithmetic on *void* pointers (no type defined)
 - 🔗 Cannot dereference (can't say, “get the value at that address” – no type defined)
- NULL is defined as 0 cast to a void * pointer
 - 🔗 #define NULL (void *) 0;

FYI: However, NULL and zero are not the same as no returned value at all, which is what is meant by a void return value (see your first C program examples)

- Is there any difference between the following two statements?
char *p=0;
char *t=NULL;
NO difference. NULL is *#defined* as 0 in the 'stdio.h' file. Thus, both *p* and *t* are NULL pointers.
- Is this a correct way for NULL pointer assignment?
int i=0;
char *q=(char*)i; // char * cannot point to an int type... even for a moment in time
NO. Correct → ***char *q=0 (or) char *q=(char*)0***
- Is the NULL pointer same as an uninitialized pointer? NO

R and L values

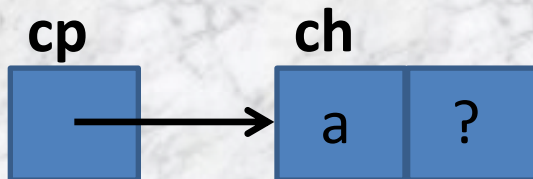
- L-value = something that can appear on the left side of an equal sign
 - 👁 A place i.e. memory location for a value to be stored
- R-value is something that can appear on the right side of an equal sign
 - 👁 A value
- Example:
 - 👁 $a = b+25$ vs $b+25 = a$
- Example:
 - 👁 `int a[30];`
 - 👁 `a[b+10]=0;`
- Example:
 - 👁 `int a, *pi;`
 - 👁 `pi = &a;`
 - 👁 `*pi = 20;`

R and L values (cont)

Given:

```
char ch = 'a';  
char *cp = &ch;
```

NOTE: the ? is the location that follows ch



Problem	Expression	R-value	L-value
1	ch	yes	yes
2	&ch	yes	illegal
3	cp	yes	yes
4	&cp	yes	illegal
5	*cp	yes	yes
6	*c+1	yes	illegal
7	*(c+1)	yes	yes
8	++cp	yes	illegal
9	cp++	yes	illegal
10	*++cp	yes	yes
11	*cp++	yes	yes
12	++*cp	yes	illegal
13	(*cp)++	yes	illegal
14	++*++cp	yes	illegal
15	++*cp++	yes	illegal

An Array of Character Pointers

```
#include<stdio.h>
int main()
{
    char *ptr1 = "Himanshu";
    char *ptr2 = "Arora";
    char *ptr3 = "TheGeekStuff";

    char* arr[3];

    arr[0] = ptr1;
    arr[1] = ptr2;
    arr[2] = ptr3;

    printf("\n [%s]\n", arr[0]);
    printf("\n [%s]\n", arr[1]);
    printf("\n [%s]\n", arr[2]);
    return 0;
}
```

// Declaring/Initializing 3 characters pointers

//Declaring an array of 3 char pointers

// Initializing the array with values

//Printing the values stored in array

Pointers to Arrays

- ▣ `<data type> (*<name of ptr>)[<an integer>]`
 - 🖱 Declares a pointer `ptr` to an array of 5 integers.
 - `int(*ptr)[5];`

```
#include<stdio.h>
int main(void)
{   char arr[3];
    char (*ptr)[3];
    arr[0] = 'a';
    arr[1] = 'b';
    arr[2] = 'c';
    ptr = &arr;
    return 0;
}
```

Declares and initializes an array 'arr' and then declares a pointer 'ptr' to an array of 3 characters. Then initializes ptr with the address of array 'arr'.

```
int *arr[8];    // An array of int pointers.
int (*arr)[8]; // A pointer to an array of integers
```


Structures

■ What is a structure?

- 🕒 One or more values, called members, with possibly dissimilar types that are stored together.
- 🕒 Used to group together different types of variables under the same name.
- 🕒 Aggregates a fixed set of labeled objects, possibly of different types, into a single object (like a record)

■ What is a structure NOT?

- 🕒 Since members are NOT the same type/size, they are not as easy to access as array elements that are the same size.
- 🕒 Structure variable names are NOT replaced with a pointer in an expression (like arrays)
- 🕒 A structure is NOT an array of its members so can NOT use subscripts.

Structure Declarations (preview)

```
struct tag {member_list} variable_list;
```

```
struct S {  
    int a;  
    float b;  
} x;
```

Declares x to be a structure having two members, a and b. In addition, the structure tag S is created for use in future declarations.

```
struct {  
    int a;  
    float b;  
} z;
```

Omitting the tag field; cannot create any more variables with the same type as z

```
struct S {  
    int a;  
    float b;  
};
```

Omitting the variable list defines the tag S for use in later declarations

```
struct S y;
```

Omitting the member list declares another structure variable y with the same type as x

```
struct S;
```

Incomplete declaration which informs the compiler that S is a structure tag to be defined later

Struct storage issues

- A struct declaration consists of a list of fields, each of which can have any type. The total storage required for a struct object is the sum of the storage requirements of all the fields, plus any internal padding.

Structure Example Preview

- This declaration introduces the type struct fraction (both words are required) as a new type.
- C uses the period (.) to access the fields in a record.
- You can copy two records of the same type using a single assignment statement, however == does not work on structs (see note link).

```
struct fraction {  
    int numerator;  
    int denominator;    // can't initialize  
};  
  
struct fraction f1, f2;    // declare two fractions  
f1.numerator = 25;  
f1.denominator = 10;  
f2 = f1;    // this copies over the whole struct
```


Structure Declarations (cont)

- So tag, member_list and variable_list are all optional, but cannot all be omitted; at least two must appear for a complete declaration.

```
struct {  
    int a;  
    char b;  
    float c;  
} x;
```

Single variable x contains 3 members

Treated different by the compiler
DIFFERENT TYPES
i.e. z = &x is ILLEGAL

So all structures of a given type must be created in a single declaration? NO.

```
struct {  
    int a;  
    char b;  
    float c;  
} y[20], *z;
```

An array of 20 structures (y); and
A pointer to a structure of this type (z)

More Structure Declarations

■ The TAG field

- 🕒 Allows a name to be given to the member list so that it can be referenced in subsequent declarations
- 🕒 Allows many declarations to use the same member list and thus create structures of the same type

```
struct SIMPLE {  
    int a;  
    char b;  
    float c;  
};
```

**Associates tag with
member list; does not
create any variables**

**So → struct SIMPLE x;
struct SIMPLE y[20], *z;**

**Now x, y, and z are all the same
kind of structure**

Typedefs → typedef <type> <name>;

■ Ex1:

■ #define true 1

■ #define false 0

■ typedef int bool;

■ bool flag = false;

■ Ex2:

■ char *ptr_to_char; // new variable

■ typedef char* ptr_to_char; // new type

■ ptr_to_char a; // new variable

Using typedefs with Structures

■ A typedef statement introduces a shorthand name for a type. The syntax is...

- 🖱️ typedef <type> <name>;
 - shorter to write
 - can simplify more complex type definitions

```
typedef struct {  
    int a;  
    char b;  
    float c;  
} Simple;
```

So → Simple x;
Simple y[20], *z;

Now x, y, and z are all the same
TYPE.

Similar to → int x;
int y[20], *z;

Typedef Structure Example

```
#include <stdio.h>
typedef struct {
    int x;
    int y;
} point;
int main(void)
{ /* Define a variable p of type point, and initialize all its members inline! */
    point p = {1,2};
    point q;
    q = p; // q.x = 1 and q.y=2
    q.x = 2;
    /* Demonstrate we have a copy and that they are now different. */
    if (p.x != q.x)
        printf("The members are not equal! %d != %d", p.x, q.x);
    return 0; }
```

Structures and Pointers

```
#include<stdio.h>

typedef struct
{ char *name;
  int number;
} TELEPHONE;

int main()
{ TELEPHONE index;
  TELEPHONE *ptr_myindex;
  ptr_myindex = &index;
  ptr_myindex->name = "Jane Doe";
  ptr_myindex->number = 12345;
  printf("Name: %s\n", ptr_myindex->name);
  printf("Telephone number: %d\n", ptr_myindex->number);
  return 0; }
```

What is going on here?

Remember:
TELEPHONE is a type
of structure;

Structures and Pointers

```
#include<stdio.h>
#include <stdlib.h>
typedef struct rec
{
    int i;
    float PI;
    char A; } RECORD;
int main()
{
    RECORD *ptr_one;
    ptr_one = (RECORD *) malloc (sizeof(RECORD));
    (*ptr_one).i = 10;
    (*ptr_one).PI = 3.14;
    (*ptr_one).A = 'a';
    printf("First value: %d\n",(*ptr_one).i);
    printf("Second value: %f\n", (*ptr_one).PI);
    printf("Third value: %c\n", (*ptr_one).A);
    free(ptr_one);
    return 0; }
```

“rec” is not necessary for given/left code, but is necessary for below code update

For below, without RECORD, warning: useless storage class specifier in empty declaration

```
struct rec *ptr_one;
ptr_one=(struct rec *) malloc (sizeof(struct rec));
ptr_one->i = 10;
ptr_one->PI = 3.14;
ptr_one->A = 'a';
printf("First value: %d\n", ptr_one->i);
printf("Second value: %f\n", ptr_one->PI);
printf("Third value: %c\n", ptr_one->A);
```

More on Structure Declarations

MEMBERS

- Any kind of variable that can be declared outside a structure may also be used as a structure member.
- Structure members can be scalars, arrays, pointers and even other structures.

ACCESS using dot operator

Two operands

- Left = name of structure variable
- Right = name of the desired member
- Result = the designated member

OPERATOR PRECEDENCE

- The subscript and dot operators have the same precedence and all associate left to right.
- The dot operator has higher precedence than the indirection

Pointer2Structure

- operator
- Left = *must* be a pointer to a structure
- Right = member

Example

- (*sp).a == sp→a
- Indirection built into arrow/infix operator
- Follow the address to the structure

```
struct COMPLEX {  
    float    f;  
    int      a[20];  
    long     *lp;  
    struct   SIMPLE s;  
    struct   SIMPLE sa[10];  
    struct   SIMPLE *sp;  
} cmplx, cmp[10];
```


Structures and Pointers

```
struct mystruct {  
    int a;  
    char* b; }; //note: could put st here instead  
struct mystruct st;  
char* pb = (char*)&st + offsetof(struct mystruct, b);
```

- `offsetof` → tells you the offset of a variable within a structure (`stddef.h`)
- should set "pb" to be a pointer to member "b" within structure "mystruct".

Self-Referential Structures

Illegal - infinite

```
struct SELF_REF {  
    int    a;  
    struct SELF_REF b;  
    int    c;  
};
```



Correction

```
struct SELF_REF {  
    int    a;  
    struct SELF_REF *b;  
    int    c;  
};
```

Watch out

```
typedef struct {  
    int    a;  
    struct SELF_REF *b;  
    int    c;  
} SELF_REF;
```



Correction

```
typedef struct SELF_REF_TAG {  
    int    a;  
    struct SELF_REF_TAG *b;  
    int    c;  
} SELF_REF;
```

Incomplete Declarations

- Structures that are mutually dependent
- As with self referential structures, at least one of the structures must refer to the other only through pointers
- So, which one gets declared first???

```
struct B;  
  
struct A {  
    struct B *partner;  
    /* etc */  
};  
  
struct B {  
    struct A *partner;  
    /* etc */  
};
```

- Declares an identifier to be a structure tag
- Use this tag in declarations where the size of the structure is not needed (pointer!)
- Needed in the member list of A

- Doesn't have to be a pointer

Initializing Structures

- Missing values cause the remaining members to get default initialization... whatever that might be!

```
typedef struct {  
    int    a;  
    char   b;  
    float  c;  
} Simple;  
  
struct INIT_EX {  
    int    a;  
    short  b[10];  
    Simple c;  
} x = { 10,  
        { 1, 2, 3, 4, 5 },  
        { 25, 'x', 1.9 }  
};
```

What goes here (hint in blue below)?

```
struct INIT_EX y = { 0, {10, 20, 30, 40, 50,  
                        60, 70, 80, 90, 100 },  
                    { 1000, 'a', 3.14 }  
                    };
```

Name all the variables and their initial values:

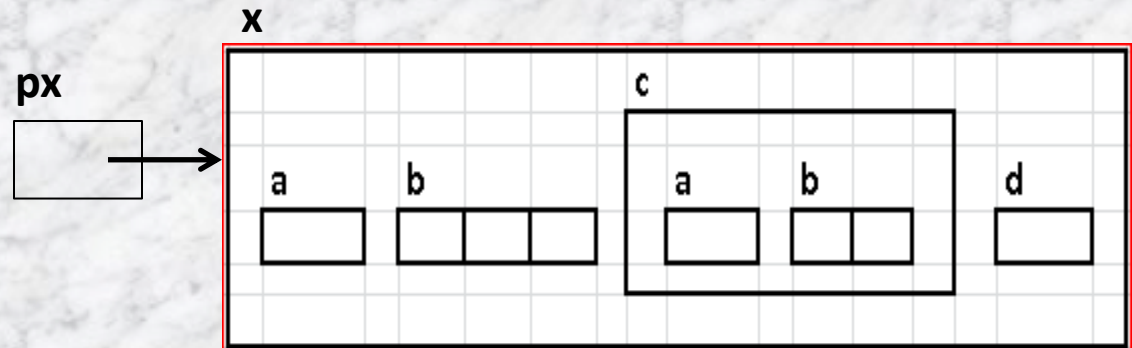
```
y.a = 0  
y.b[0] = 10; y.b[1] = 20; y.b[2] = 30; etc  
y.c.a = 1000;  
y.c.b = 'a';  
y.c.c = 3.14;
```


Structure memory (again)

What does memory look like?

```
typedef struct {  
    int    a;  
    short  b[2];  
} Ex2;
```

```
typedef struct EX {  
    int    a;  
    char   b[3];  
    Ex2    c;  
    struct EX *d;  
} Ex;
```



Given the following declaration, fill in the above memory locations:

```
Ex x = { 10, "Hi", { 5, { -1, 25 } }, 0 };
```

```
Ex *px = &x;
```

Structures as Function arguments

- Legal to pass a structure to a function similar to any other variable but often inefficient

```
/* electronic cash register individual
transaction receipt */
#define PRODUCT_SIZE 20;
typedef struct {
    char    product[PRODUCT_SIZE];
    int     qty;
    float   unit_price;
    float   total_amount;
} Transaction;
```

- Function call:**
 - `print_receipt(current_trans);`
 - Copy by value copies 32 bytes to the stack which can then be discarded later
- Instead...**
 - `(Transaction *trans)`
 - `trans->product // fyi: (*trans).product`
 - `trans->qty`
 - `trans->unit_price`
 - `trans->total_amount`
 - `print_receipt(¤t_trans);`
 - `void print_receipt(Transaction *trans)`

```
void print_receipt (Transaction trans) {
    printf("%s\n", trans.product);
    printf("%d @ %.2f total %.2f\n", trans.qty, trans.unit_price, trans.total_amount);
}
```

Dynamic Memory Allocation (again?!)

- Dynamic allocation allows a program to create space for a structure whose size isn't known until runtime.
 - 🕒 memory is more explicitly (but more flexibly) managed, typically, by allocating it from the *heap*, an area of memory structured for this purpose.
- The ***malloc*** and ***calloc*** functions both allocate memory and return a **void** pointer to it; NULL is returned if the requested allocation could not be performed (in `stdlib.h`)... MUST check for this!
 - 🕒 ***malloc***
 - Argument: # of bytes needed
 - Leaves the memory uninitialized
 - 🕒 ***calloc***
 - Arguments: number of elements AND the size of each element
 - Initializes the memory to zero before returning
- The ***free*** function
 - 🕒 You may not pass a pointer to this function that was not obtained from an earlier call to `malloc/calloc`.
 - 🕒 Memory must not be accessed after it has been freed.
- Memory Leaks
 - 🕒 Memory that has been dynamically allocated but has not been freed and is no longer in use.
 - 🕒 Negative because it increases the size of the program and lead to problems.

DMA Example

Set each element of the newly allocated integer array of five elements to zero instead of declaring `int_array[5]`

```
int *pi_save, *pi;
pi = malloc(20);

if (pi == NULL)
{
    printf("Out of memory!\n");
    exit(1);
}

for (int x = 0; x < 5; x +=1)
    *pi++ = 0;

// print
```

QUESTIONS

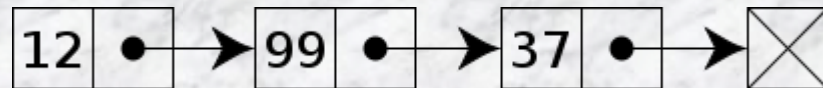
1. What are the values in the new memory before initializing to zero?
2. Where is `pi` pointing to after the for loop?
3. What does the print loop look like?
4. How update to use `calloc`?
5. How free the memory?

(see `dma1.c`)

Linked List Node structure

- A linked list is...a data structure consisting of a group of nodes which together represent a sequence
- Simply, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence
- Allows for efficient insertion or removal of elements from any position in the sequence (vs an array).
- Data items need not be stored contiguously in memory
- Major Disadvantage:
 - 👁️ does not allow random access to the data or any form of efficient indexing

```
/* Node Structure */  
struct node {  
    int data;  
    struct node *next; }  
}
```



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

DMA structure (linked list)

```
struct myRecord {  
    char firstName[20];  
    char lastName[25];  
    int employeeID;  
    struct myRecord * nextRecord;  
};
```

```
// point to first structure in the list  
struct myRecord *headPtr;
```

```
headPtr = (struct myRecord *) malloc(sizeof(myRecord));  
// when allocate another structure,  
// the pointer returned should be assigned to the first record's pointer
```

(see linklst1.c) → push and print

Redirection File I/O

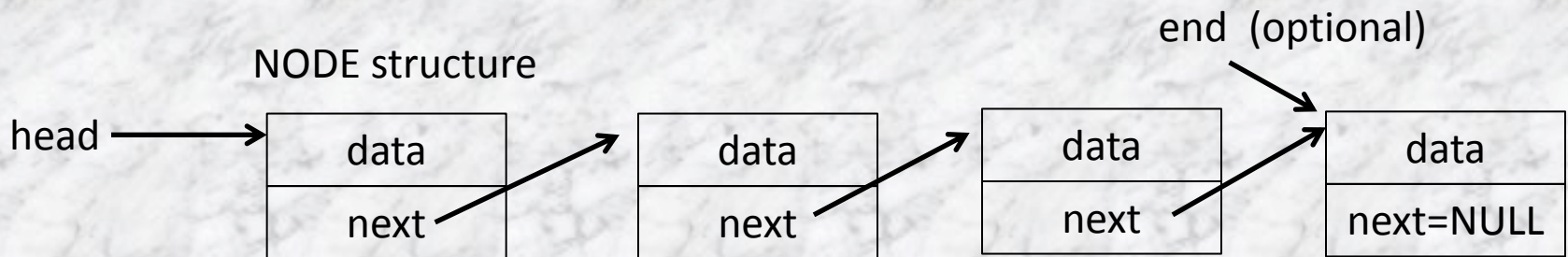
- Part of the operating system (linux)
- % lab2p2file < lab2p2in >! lab2p2out
- ! overwrites if the file already exists

```
input = 0;
scanf(..., input);
while (input != 0)
{ loop stuff...
  input = 0;
  scanf(..., input);
}
```

Input File:

```
War_Eagle!
How_many_WORDS_workhere?
i
$shake_u_r_booty:)_!
Og_sbuCk!!!
REALLY_really_really_really___really_long??!!_
Hi.01234_How_R_U_?
|
```

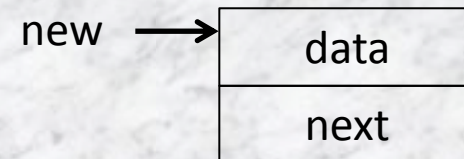
Linked List setup



NEED TO:

Allocate a new node structure with DMA

Add information to data section



Linked List ADD/DELETE node

OPERATION



FRONT



END



MIDDLE

ADD

new->next = head
head = new

ptr->next = new
new->next = NULL

new->next = ptr->next
ptr->next = new

SEARCH

```
found = false;
ptr = head;
while(ptr != NULL)           //what if nothing in list?
{   if(ptr->data == val)     // found what searching for
    {   found = true;
        break; }
    else { ptr = ptr->next; }
}
```

// if found still false, didn't find

DELETE
(fyi: free ptr)

head = ptr->next
// what if only node?

//if ptr->next=NULL
prev = ptr
prev->next = NULL

prev->next = ptr->next

Linked List operations

- Initialize the list
- Push/Insert a value onto the list
- Search the list
- Pop/Remove a value off of the list
- Print the list

```
void InitList(struct list *sList);  
  
/* Initializes the list structure */  
void InitList(struct list *sList) {  
    sList->start = NULL; }  
}
```

```
void push(struct list *sList, int data);  
  
/* Adds a value to the front of the list */  
void push(struct list *sList, int data) {  
    struct node *p;  
    p = malloc(sizeof(struct node));  
    p->data = data;  
    p->next = sList->start;  
    sList->start = p; }  
}
```

```
void pop(struct list *sList)  
  
/* Removes the first value of the list */  
void pop(struct list *sList) {  
    if(sList->start != NULL) {  
        struct node *p = sList->start;  
        sList->start = sList->start->next;  
        free(p); } }  
}
```

(see linklst2.c)

Bitwise Operations

- Many situation, need to operate on the bits of a data word –
 - Register inputs or outputs
 - Controlling attached devices
 - Obtaining status
- Corresponding bits of both operands are combined by the usual *logic operations*.
- Apply to all kinds of *integer* types
 - 🖱 Signed and unsigned
 - 🖱 char, short, int, long, long long

Bitwise Operations (cont)

- **& – AND**

- Result is **1** if both operand bits are **1**

- **| – OR**

- Result is **1** if either operand bit is **1**

- **^ – Exclusive OR**

- Result is **1** if operand bits are different

- **~ – Complement**

- Each bit is reversed

- **<< – Shift left**

- Multiply by 2

- **>> – Shift right**

- Divide by 2

Examples

a

1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---

b

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

NOTE: when signed → all the same

FYI: integers are really 32 bits so what is the “real” value?

~a has preceding 1's and a<<2 is 0x 3c0

```
unsigned int c, a, b;
```

```
c = a & b;           // 1010 0000
c = a | b;           // 1111 1010
c = a ^ b;           // 0101 1010
c = ~a                // 0000 1111
c = a << 2;          // 1100 0000
c = a >> 3;          // 0001 1110
```

Bitwise AND/OR

char x = 'A';
tolower(x) returns 'a'... HOW?

char y = 'a';
toupper(y) returns 'A'... HOW?

'A' = 0x41 = 0100 0001

'a' = 0x61 = 0110 0001

"mask" = 0010 0000
Use OR

'A' = 0100 0001
mask = 0010 0000 |
'a' 0110 0001

"mask" = 1101 1111
Use AND

'a' = 0110 0001
mask = 1101 1111 &
'A' 0100 0001

Notice the masks are complements of each other
TRY: char digit to a numeric digit

Bitwise XOR

- The bitwise XOR may be used to invert selected bits in a register (toggle)
- XOR as a short-cut to setting the value of a register to zero

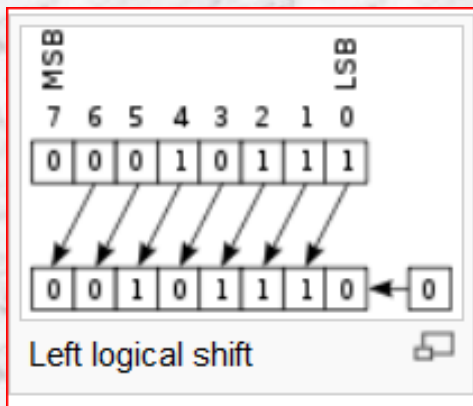
0100 0010

0000 1010 XOR (toggle)

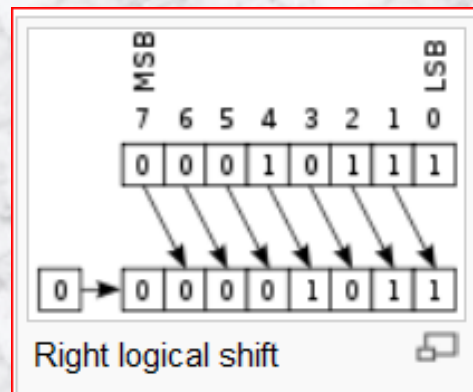
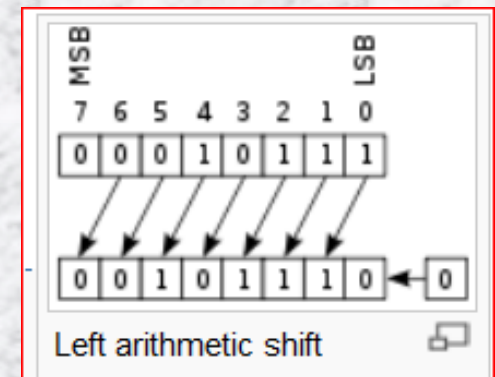
0100 1000

Bitwise left/right shifts

- Possible overflow issues
- Exact behavior is implementation dependent

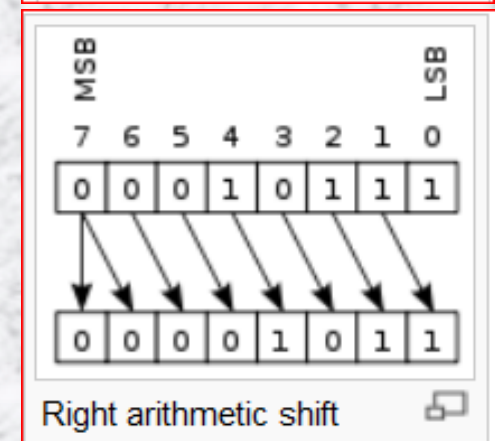


When you shift left by k bits ==
multiplying by 2^k



When you shift right by k bits ==
dividing by 2^k

***** If it's signed, then it's***
implementation dependent.**



Bitwise right shifts

a

1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

 b

0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
unsigned int c, a;
```

```
  c = a >> 3;    c 

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|


```

```
signed int c, a, b;
```

```
  c = b >> 3;    c 

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|



|   |   |   |
|---|---|---|
| 1 | 0 | 1 |
|---|---|---|


```

```
  c = a >> 3;    c 

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|


```

EXAMPLE: 8-bit instruction format

101 01000 // ADD 8 → ALU adds ACC reg to value at address 8

To get just the instruction i.e. 101... shift right by 5


To get just the address i.e. 01001... shift left by 3, then right by 3

C example...

```
#include <stdio.h>
void main()
{
    signed int c, d, a, b, e, f;
    a = 0xF0F0;
    b = 0x5555;
    e = 0b01000001;
    f = 'A';

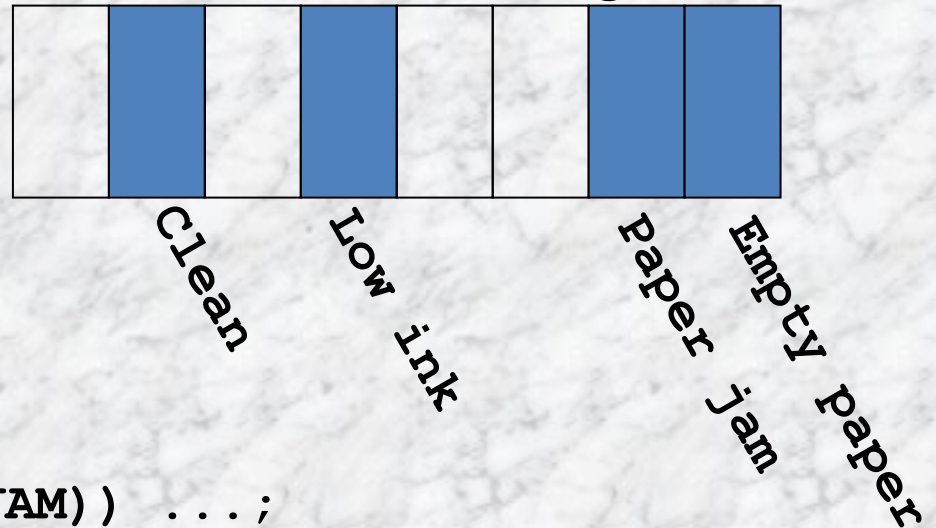
    c = b >> 3;
    d = a >> 3;

    printf("b >> 3 is %x\n",c);
    printf("a >> 3 is %x\n",d);
    printf("binary = %x\n",e);
    printf("char a = %c",f);
}
```

 Output is:
b >> 3 is aaa
a >> 3 is 1e1e
binary = 41
char a = A

Traditional Bit Definition

8-bit Printer Status Register



```
#define EMPTY    01
#define JAM      02
#define LOW_INK  16
#define CLEAN    64
```

```
char status;
if (status == (EMPTY | JAM)) ...;
if (status == EMPTY || status == JAM) ...;
while (! status & LOW_INK) ...;
```

```
int flags |= CLEAN    /* turns on CLEAN bit */
int flags &= ~JAM     /* turns off JAM bit */
```

Traditional Bit Definitions

- Used very widely in *C*
 - Including a *lot* of existing code
- No checking
 - You are on your own to be sure the right bits are set
- Machine dependent
 - Need to know *bit order* in bytes, *byte order* in words
- Integer fields within a register
 - Need to **AND** and shift to extract
 - Need to shift and **OR** to insert

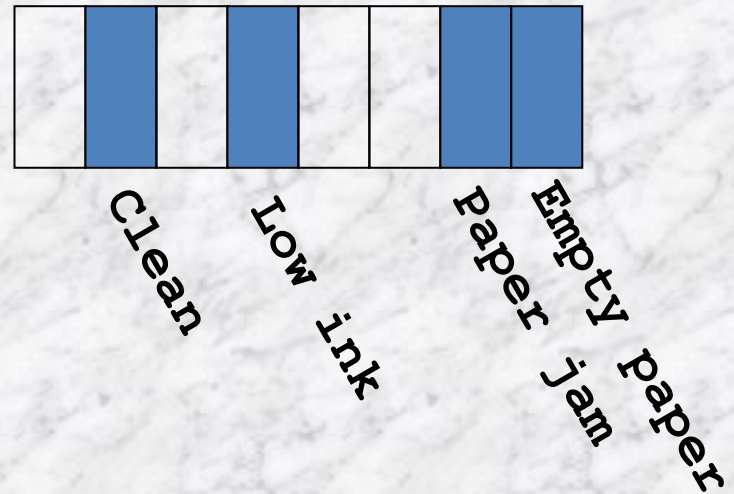
Modern Bit-field Definitions

```
struct statusReg {  
    unsigned int empty      :1;  
    unsigned int jam        :1;  
    unsigned int lowInk     :2; //???  
    unsigned int needsCleaning :1; //???  
    unsigned int             :1; //???  
};
```

```
struct statusReg s;
```

```
if (s.empty && s.jam) ...;  
while(! s.lowInk) ...;
```

```
s.needsCleaning = true;  
s.Jam = false;
```



Conditional Operator

- Consists of two symbols
 - 🖱 Question mark
 - 🖱 Colon
- Syntax: $\text{exp1} ? \text{exp2} : \text{exp3}$
- 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!= '10')
- Exchanging values:
 - 👁 t=x, x=y, y=t;

File function summary

■ Open/Close files

🖱️ `fopen()` – open a stream for a file

🖱️ `fclose()` – closes a stream

■ One character at a time:

🖱️ `fgetc()` – similar to `getchar()`

🖱️ `fputc()` – similar to `putchar()`

■ One line at a time:

🖱️ `fprintf()/fputs()` – similar to `printf()`

🖱️ `fscanf()/fgets()` – similar to `scanf()`

■ File errors

🖱️ `perror()` – reports an error in a system call

Text Streams

- Files accessed through the FILE mechanism provided by `<stdio.h>`
 - http://www.acm.uiuc.edu/webmonkeys/book/c_guide/2.12.html
- Text streams are composed of lines.
- Each line has zero or more characters and are terminated by a new-line character which is the last character in a line.
- Conversions may occur on text streams during input and output.
- Text streams consist of only printable characters, the tab character, and the new-line character.
- Spaces cannot appear before a newline character, although it is implementation-defined whether or not reading a text stream removes these spaces.

File constants <stdio.h>

- FILE – a variable type suitable for string information for a file stream
- fpos_t – a variable type suitable for starting any position in a file
- NULL – value of a null pointer constant
- EOF – negative integer which indicates end-of-file has been reached
- FOPEN_MAX – integer which represents the maximum number of files that the system can guarantee that can be opened simultaneously
- FILENAME_MAX – integer which represents the longest length of a char array
- stderr/stdin/stdout – pointers to FILE types which correspond to the standard streams

File usage

- When a program begins, there are already three available streams which are predefined and need not be opened explicitly and are of type “pointer to FILE”
 - standard input
 - standard output
 - standard error
- Files are associated with streams and must be opened to be used.
- The point of I/O within a file is determined by the file position.
- When a file is opened, the file position points to the beginning of the file (unless the file is opened for an append operation in which case the position points to the end of the file).
- The file position follows read and write operations to indicate where the next operation will occur.
- When a file is closed, no more actions can be taken on it until it is opened again.
- Exiting from the main function causes all open files to be closed.

Open/Read a File – one char at a time

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    char ch;
    FILE *fp;
    fp = fopen("lab2p2in", "r"); // read mode
    if( fp == NULL ) {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE); }
    printf("The contents of the file is :- \n\n");
    while( ( ch = fgetc(fp) ) != EOF )
        printf("%c",ch);
    fclose(fp);
    return 0; }
```

C programming code to open a file and print its contents to the screen, one character at a time.

//fileio1.c

- (1) fgetc returns the value of an int that is converted from the character
- (2) What happens if delete lab2p2in file? i.e. it can't be found to open?

File open and close

- **FILE *fopen(const char *filename, const char *mode);**
- **Mode... (lots more!)**
 - 🕒 r – read text mode
 - 🕒 w – write text mode (truncates file to zero length or creates a new file)
 - 🕒 If the file does not exist and it is opened with read mode (r), then the open fails → need to check for this
- **Declaration: int fclose(FILE *stream);**
 - 🕒 Closes the stream.
 - 🕒 If successful, it returns zero.
 - 🕒 On error it returns **EOF**.
- **perror**
 - 🕒 **void perror(const char *str);**
 - 🕒 Prints a descriptive error message to stderr. First the string *str* is printed followed by a colon then a space (your error message). Then an error message based on the current setting of the variable **errno** is printed (system error message).

fgetc and fputc

- Declaration: **int fgetc(FILE *stream);**
 - 🕒 Gets the next character (an **unsigned char**) from the specified stream and advances the position indicator for the stream.
 - 🕒 On success the character is returned.
 - 🕒 If the end-of-file is encountered, then **EOF** is returned and the end-of-file indicator is set.
 - 🕒 If an error occurs then the error indicator for the stream is set and **EOF** is returned.
- Declaration: **int fputc(int char, FILE *stream);**
 - 🕒 Writes a character (an **unsigned char**) specified by the argument *char* to the specified stream and advances the position indicator for the stream.
 - 🕒 On success the character is returned.
 - 🕒 If an error occurs, the error indicator for the stream is set and **EOF** is returned.

Open/Read/Write/Close... one char at a time

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    char ch, chout;
    FILE *fpin, *fpout;
    fpin = fopen("lab2p2in","r"); // read mode
    fpout = fopen("lab2p2inout","w"); // write mode
    if( fpin == NULL ) {
        perror("Error while opening the input file.\n");
        exit(EXIT_FAILURE); }
    if (fpout == NULL ) {
        perror("Error while opening the output file.\n");
        exit(EXIT_FAILURE); }
    while( ( ch = fgetc(fpin) ) != EOF && chout != EOF )
        chout = fputc(ch,fpout); // ret char if success ow EOF
    fclose(fpin);
    fclose(fpout);
    return 0; }
```

C programming code to open a file and print its contents to the another file, one character at a time.

`//fileio2.c`

Lab2p2 excerpt example

```
FILE *infp, *outfp;
char * mode = "r";
char outfile[] = "lab2p2out";

char input[101], save_first_letter;
char *inptr;
int first_letter = TRUE, n=101;

infp = fopen("lab2p2in","r");
if (infp == NULL){
    fprintf(stderr, "can't open input file lab2p2in!\n");
    exit(EXIT_FAILURE); }

outfp = fopen(outfile,"w");
if (outfp == NULL) {
    fprintf(stderr, "Can't open output file %s!\n", outfile);
    exit(EXIT_FAILURE); }

fgets(input,n,infp);
while (!feof(infp))
    { // etc
        fgets(input,n,infp);
    }
//close files
```

- `fgets(buffer,size,stdin);`
- `buffer` is the location of your string storage space or buffer.
- `size` is the number of characters to input. This sets a limit on input
- Note that `fgets()` also **reads in the carriage return** (enter key; newline character) at the end of the line. That character becomes part of the string you input.
- `fscanf(infp,"%s",input);`
- **`while (!feof(infp))`**

fgets vs fscanf

- Declaration: **char *fgets(char *str, int n, FILE *stream);**
 - 🕒 Reads a line from the specified stream and stores it into the string pointed to by *str*.
 - 🕒 It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
 - 🕒 **The newline character is copied to the string.**
 - 🕒 A null character is appended to the end of the string.
 - 🕒 On error a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.
- Declaration: **int fscanf(FILE *stream, const char *format, ...);**
 - 🕒 Reading an input field (designated with a conversion specifier) ends when an incompatible character is met, or the width field is satisfied.
 - 🕒 On success the number of input fields converted and stored are returned. If an input failure occurred, then EOF is returned.
 - 🕒 Returns EOF in case of errors or if it reaches eof

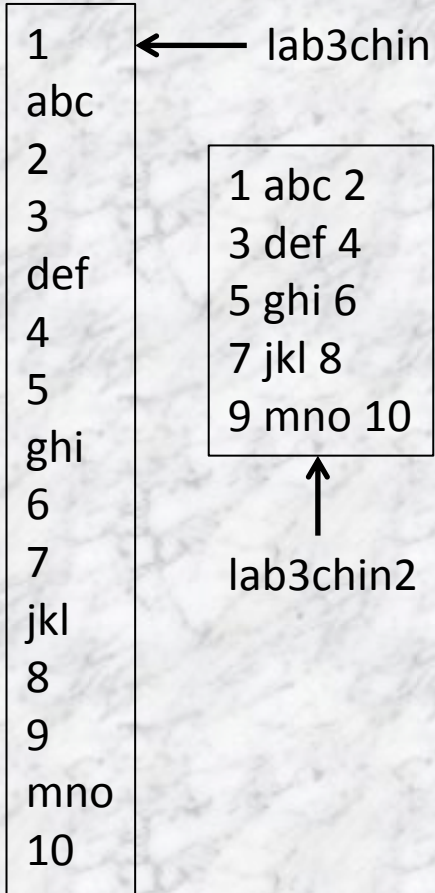
fprintf and feof

- Declaration:
 - 🕒 `int fprintf(FILE *stream, const char *format, ...);`
 - 🕒 sends formatted output to a stream
 - 🕒 Just like `printf`, but puts file pointer as first argument
 - 🕒 In lab2p2:
 - `fprintf(outfp, "Your sipher coded message is %s\n",input);`
- Declaration: `int feof(FILE *stream);`
 - 🕒 Tests the end-of-file indicator for the given stream
 - 🕒 If the stream is at the end-of-file, then it returns a nonzero value. If it is not at the end of the file, then it returns zero.

Lab3 fileio example



See handout



Enumerated Data Types

```
#include <stdio.h>
main() {
    char *pwest="west",*pnorth="north", *peast="east", *psouth="south";
    enum location { east=1, west=2, south=3, north=4};
    enum location direction;
    direction = east;
    if( direction == east )
        printf("Cannot go %s\n", peast); }

    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;
    this_month = feb;
    printf("What month is it? %d\n",this_month);
}
```


Enumerated Data Types explained

- An enumerated type is one whose values are symbolic constant rather than literal.
- Declaration example:
 - 🕒 `enum Jar_Type {CUP, PINT, QUART, HALF_GALLON, GALLON};`
- The above example declares a type called Jar_Type
- Variables of this type are declared like this:
 - 🕒 `enum Jar_Type milk_jug, gas_can, medicine_bottle;`
- If there is only one declaration of variables of a particular enumerated type (i.e. no type name), both statements may be combined:
 - 🕒 `enum { CUP, PINT, QUARTER, HALF_GALLON, GALLON} milk_jug, gas_can, medicine_bottle;`

Enumerated Data Type (cont)

- Variables declared with an enumerated type are actually stored as integers.
- Internally, the symbolic names are treated as integer constants
 - 👁 By default, CUP =0, PINT=1, QUART=2, etc.
- Caution: don't mix them indiscriminately with integers – even though it is viable.
 - 👁 milk_jug = -623;
 - 👁 int a = PINT;
- The variable cannot be assigned any values outside those specified in the initialization list for the declaration of the enum type

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.

Pointers and Constants


 `int *pi;`

 pointer to int

 `int const *pci;`

 point to constant int

 `int *const cpi;`

 constant pointer to int

 `int const *const cpci;`

 constant point to constant int

Storage Class

- Refers to the type of memory in which the variable's value is stored which in turn defines different characters for the variable
 - 🕒 Ordinary memory
 - 🕒 Runtime stack
 - 🕒 Hardware registers
- Determines when the variable is created and destroyed and how long it will retain its value
- Introduction
 - 🕒 Auto → automatic
 - Local variable known only to the function in which it is declared; default storage class; stored in RAM (i.e. on the stack).
 - 🕒 Static
 - 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.
 - 🕒 Register
 - Local variables that are stored in the CPU memory register; must be initialized explicitly;
- The default storage class for a variable depends on where it is declared

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

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

Storage Classes – static (cont)

- 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

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 (long story – previous values? will revisit; note: not allowed to take the address of a register variable)

Identifier Storage Class Summary

- Designated before the type
- 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;
}
```

Nm/Ty	STORAGE	LINKAGE	SCOPE & INITIAL VALUE
a = int	static	external	accessible to x but not y with init value = 1
b = char	static	none	accessible to x and y with init value = 2
c = int	automatic	none	local to x with init value = 3
d = float	static	none	local to x with init value = 4

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 to 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.

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

Block scope

- 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)

File scope

- 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

```
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
Prototype scope = 3,8
Block = **6**, 7, **9**, 10 (**f?**)
5 = block scope in function body
? What if 6 had included an e?

ScopeExample.docx

Name (Line)	Storage Class	Scope	Linkage	Initial Value
w (1)	static	1–8, 17–31	internal	5
x (2)	static	2–18, 23–31	external	Note a
func1 (4)	–	4–31	external	–
a (4)	auto	5–18, 23	none	Note b
b, c (4)	auto	5–11, 16–23	none	Note b
d (6)	auto	6–8, 17, 23	none	garbage
e (6)	auto	6–8, 17–23	none	1
d (9)	auto	9–11, 16	none	garbage
e, w (9)	auto	9–16	none	garbage
b, c, d (12)	auto	12–15	none	garbage
y (13)	static	13–15	none	2
a, d, x (19)	register	19–22	none	garbage
y (20)	static	20–22	external	Note a
y (24)	static	24–31	internal	zero
func2 (26)	–	26–31	external	–
a (26)	auto	27–31	none	Note b
y (28)	static	28–31	Note c	see y (24)
z (29)	static	29–31	none	zero

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.

SUMMARY

Variable Type	Where Declared	Stored on Stack	Scope	If Declared static
global	outside of all blocks	no (1)	remainder of this source file	prevents access from other source files
local	beginning of a block	yes (2)	throughout the block (3)	variable is not stored on the stack, keeps its value for the entire duration of the program
formal parameters	function header	yes (2)	throughout the function (3)	not allowed
(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 <i>*not*</i> 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				

Int main(with arguments)

Options:

- int main(void);
- int main();
- int main(int argc, char **argv);
- int main(int argc, char *argv[]);

```
myFilet p1 p2 p3
```

results in something like:

m	y	F	i	l	e	\0	p	1	\0	p	2	\0	p	3	\0
argv[0]							argv[1]			argv[2]		argv[3]			

- The parameters given on a command line are passed to a C program with two predefined variables
 - The count of the command-line arguments in argc
 - The individual arguments as a character strings in the pointer array argv
 - The names of argc and argv may be any valid identifier in C, but it is common convention to use these names
- But wait... There is no guarantee that the strings are stored as a contiguous group (per normal arrays)

int main(argc, *argv[])

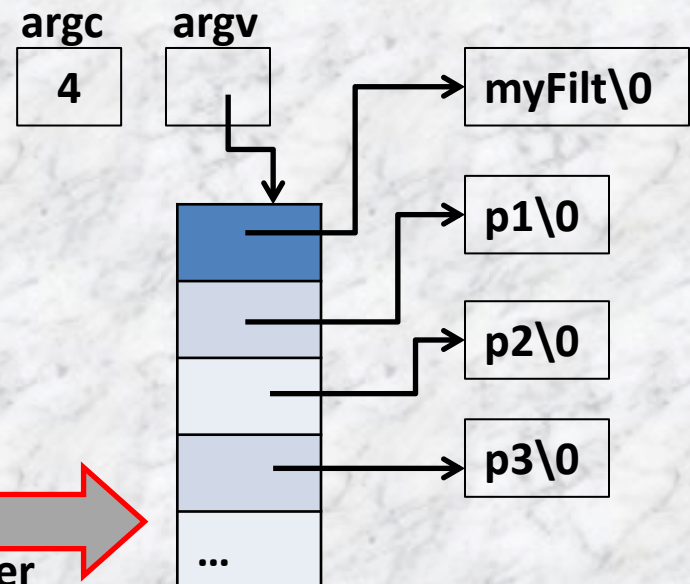
- The name of the program, argv[0], may be useful when printing diagnostic messages
- The individual values of the parameters can be accessed:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    printf ("argc\t= %d\n", argc);
    for (i = 0; i < argc; i++)
        printf ("argv[%i]\t= %s\n", i, argv[i]);
    return 0;
}
```

***argv[] also seen as **argv**



Array of pointers where each element is a pointer to a character

Command Line Arguments

- It is guaranteed that `argc` is non-negative and that `argv[argc]` is a null pointer.
- By convention, the command-line arguments specified by `argc` and `argv` include the name of the program as the first element if `argc` is greater than 0
- For example, if a user types a command of "rm file", the shell will initialize the `rm` process with `argc = 2` and `argv = ["rm", "file", NULL?]`
- The `main()` function is special; normally every C program must define it exactly once.

CLA - Example 1

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    if ( argc != 3)
        printf("Usage:\n %s Integer1 Integer2\n",argv[0]);
    else
        // ascii to integer
        printf("%s + %s = %d\n",argv[1],argv[2], atoi(argv[1])+atoi(argv[2]));
    return 0;
}
```

CLA – Example 2

```
#include <stdio.h>
main( int argc, char *argv[])
{ FILE *in_file, *out_file, *fopen();
  int c;
  if( argc != 3 ) {
    printf("Incorrect, format is FCOPY source dest\n");
    exit(2);  }
  in_file = fopen( argv[1], "r");
  if( in_file == NULL )
    printf("Cannot open %s for reading\n", argv[1]);
  else { out_file = fopen( argv[2], "w");
    if ( out_file == NULL )
      printf("Cannot open %s for writing\n", argv[2]);
    else { printf("File copy program, copying %s to %s\n", argv[1], argv[2]);
      while ( (c=getc( in_file ) ) != EOF )
        putc( c, out_file );
      putc( c, out_file); /* copy EOF */
      printf("File has been copied.\n"); fclose( out_file); } fclose( in_file); } }
```

Rewrite the program which copies files, ie, FCOPY.C to accept the source and destination filenames from the command line. Include a check on the number of arguments passed.

Header and Makefile example

The image shows a multi-window Emacs editor environment. On the left, three source files are open: `mkfact.c`, `mkfunc.h`, and `mkhello.c`. The `mkfact.c` window shows a recursive factorial function. The `mkfunc.h` window shows a `print_hello()` function. The `mkhello.c` window shows a `main` function that calls `print_hello()` and `mkfact(5)`. On the right, a `makefile` window shows the build rules for `mkhello`, `mkmain.o`, `mkfact.o`, and `mkhello.o`. Below the source files, a terminal window shows the execution of `make clean`, `make`, and `make` again. The output of the first `make` command shows the compilation of the source files and the execution of the resulting `mkhello` binary, which prints "Hello World!" and "The factorial of 5 is 120". The second `make` command shows that there is nothing to be done for 'all'. A red box with the text "Start here" is positioned over the terminal window, pointing to the first `make` command.

```
emac: mkfact.c
File Edit View Cmds Tools Options Buffers C Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace
#include "mkfunc.h"

int mkfact(int n){
  if(n!=1){
    return(n * mkfact(n-1));
  }
  else return 1;
}

-----XEmacs: mkfact.c (C PenDel Font Abbrev)-

emac: mkfunc.h
File Edit View Cmds Tools Options Buffers Resolve/C++
Open Dired Save Print Cut Copy Paste Undo Spell Replace
void print_hello();
int mkfact(int n);

-----XEmacs: mkfunc.h (C: PenDel Font Abbrev)-

emac: mkhello.c
File Edit View Cmds Tools Options Buffers C Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace
#include "mkfunc.h"

void print_hello(){
  printf("Hello World!\n");
}

-----XEmacs: mkhello.c (C PenDel Font Abbrev)-

emac: makefile
File Edit View Cmds Tools Options Buffers Makefile Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail In
all: mkhello

mkhello: mkmain.o mkfact.o mkhello.o
  gcc mkmain.o mkfact.o mkhello.o -o mkhello

mkmain.o: mkmain.c
  gcc -c mkmain.c

mkfact.o: mkfact.c
  gcc -c mkfact.c

mkhello.o: mkhello.c
  gcc -c mkhello.c

clean:
  rm -rf *.o mkhello

-----XEmacs: makefile
Wrote /home/5/reeves/cse2421

/dev/pts/22@alpha
File Edit View Search Terminal Help
% make clean
rm -rf *.o mkhello

/home/5/reeves/cse2421
% make
gcc -c mkmain.c
gcc -c mkfact.c
gcc -c mkhello.c
gcc mkmain.o mkfact.o mkhello.o -o mkhello

/home/5/reeves/cse2421
% mkhello
Hello World!
The factorial of 5 is 120
/home/5/reeves/cse2421
% make
make: Nothing to be done for `all'.

/home/5/reeves/cse2421
%

emac: mkmain.c
File Edit View Cmds Tools Options Buffers Makefile Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail In
#include <stdio.h>
#include "mkfunc.h"

int main(){
  print_hello();
  printf("The factorial of 5 is %d", mkfact(5));

  return 0;
}

-----XEmacs: mkmain.c (C PenDel Font Abbrev)-----L1--
Gnuserv process exited; restart with 'M-x gnuserv-start'
```

What is happening?

- The `-c` option on the `gcc` command only *compiles* the files listed
- Once all 3 C files are correctly compiled, then using `gcc` with the `-o` option allows object files (notice the `.o` extensions) to be merged into one executable file.
- Notice where all “`mkfunc.h`” is included

Library includes

- The compiler supports two different types of #includes
 - 🖱 Library files
 - 🖱 Local files
- #include <filename>
- #include "filename"
- By convention, the names of the standard library header files end with a .h suffix
 - 🖱 Where? → /usr/include

Creating header files

- In our case, be sure to save your header file in a ‘directory where you are going to save the program’ (NOTE: This is important. Both the header file and the program must be in the same directory, if not the program will not be able to detect your header file).
- The header file cannot be included by
 - 👁 `#include <headerfilename.h>`
- The only way to include the header file is to treat the filename in the same way you treat a string.
 - 👁 `#include "headerfilename.h"`

Makefile Overview

- Makefiles are a UNIX thing, not a programming language thing
- Makefiles contain UNIX commands and will run them in a specified sequence.
- You name of your makefile has to be: makefile or Makefile
- The directory you put the makefile in matters!
- You can only have one makefile per directory.
- Anything that can be entered at the UNIX command prompt can be in the makefile.
- Each command must be preceded by a TAB and is immediately followed by a carriage return
- MAKEFILES ARE UNFORGIVING WHEN IT COMES TO WHITESPACE!
- To execute... must be in the directory where the makefile is:
 - 👁 % make tag-name (also called section name)

Makefile Details

- Compiling our example would look like:
 - 🖱 `gcc -o mkhello mkmain.c mkhello.c mkfact.c`
 - 🖱 OR
 - 🖱 `gcc mkmain.c mkhello.c mkfact.c -o mkhello`
- The basic makefile is composed of lines:
 - 🖱 *target: dependencies [tab] system command*
 - 🖱 “all” is the default target for makefiles
 - `all: gcc -o mkhello mkmain.c mkhello.c mkfact.c`
 - 🖱 The *make* utility will execute this target, “all”, if no other one is specified.

Makefile dependencies

- Useful to use different targets
 - 🕒 Because if you modify a single project, you don't have to recompile everything, only what you modified

- In the class example of the makefile:
 - 🕒 All has only dependencies, no system commands
 - 🕒 If order for make to execute correctly, it has to meet all the dependencies of the called target (i.e. in this case all)
 - 🕒 Each of the dependencies are searched through all the targets available and executed if found.
 - 🕒 make clean
 - Fast way to get rid of all the object and executable files (to free disk space)
 - -f do not prompt
 - -r remove directories and their contents recursively