C

CSC 352, Fall 2015
The University of Arizona
William H. Mitchell
whm@cs

A Little History

A little history

In mid-1969 Ken Thompson at Bell Laboratories writes the first version of the UNIX operating system in assembler on a Digital Equipment Corporation PDP-7.

Thompson starts to write a FORTRAN compiler but changes direction and creates a language called "B", a cut-down version of BCPL. ("Basic Combined Programming Language")

B was interpreted and was oriented towards untyped word-sized objects. Dennis Ritchie adds types and writes a compiler. Ritchie initially calls it NB for New B, but renames it C. (1972)

In 1973 the Third Edition of UNIX is released. It is largely in C.

In 1978 *The C Programming Language* by Brian Kernighan and Dennis Ritchie is published.

A little history, continued

A standardization effort began in 1983 and culminated in a standard designated as ANSI/ISO 9899:1990. This standard was long known as "ANSI C" and is now referred to as "C89" or "C90".

Appendix C in King talks about C89 vs. "K&R C".

The first public release of C++ was in 1985. C++ started attracting C developers immediately. By the late 1980s there was a substantial and steady migration of developers from C to C++.

In 1999 a new standard, ISO/IEC 9899:1999, was finalized. It is known as C99. With the C99 standard, C is no longer a subset of C++.

Appendix B in King talks about C99 vs. C89

The latest standard is ISO/IEC 9899:2011and is known as C11. As it was being developed it was first called C0X (zero) and then C1X.

For our purposes there are few practical differences between C99 and C11.

Current plan: We'll try to be as close to C11 as is practical.

`fall15/{INCITS+ISO+IEC+9899-201x.pdf,c11.pdf}` is a freely available, all-but-final copy of the C11 standard.

The Big Picture with C

What is C and Where is it Used?

"C is a general-purpose programming language which features economy of expression, modern control flow, and a rich set of operators."
—Kernighan & Ritchie, 1978

There are large production systems written in C in virtually every software application area.

C is most commonly used in "systems software" such as operating systems, graphics libraries, editors, compilers, interpreters, virtual machines, and a wide range of utilities.

C is a common choice for the "embedded" software that is used to control just about every complex electro-mechanical system on the planet (and off), ranging from toasters to spacecraft.

What's great about C?

C was designed to be "close to the machine"—there's usually a direct mapping between an expression in C and the corresponding machine code.

"What you write is what you get."

C makes very efficient use of memory. An N-character string occupies N+1 bytes of memory. An array of twenty 32-bit integers occupies 80 bytes of memory.

C programs typically run 80-90% as fast as an equivalent program hand-coded in assembler. On some architectures with complex instruction scheduling rules, a good C compiler generates better code than humans.

C has very few constraints—it's hard to find something you simply can't write in C.

In some cases, compilers for other languages generate C source code rather than assembly code. ("C as assembler.") That language can then be made available quickly on any machine that has a (good) C compiler.

With a little effort, a C program is portable from one architecture to another. And, lots of machines have C compilers.

What's not so great about C?

With respect to many languages, C has quite a few shortcomings:

- No abstract types such as strings and lists.
- Flat namespace for functions.
- No protection against out-of-bounds array references.
- No garbage collection—memory in the heap must be managed by hand.
- Bugs often manifest themselves at times and in places that are far removed from the faulty code.

It's not hard to get into trouble with C. A fact:

> Bad C code is the cause of many security vulnerabilities. It's a pretty good bet that a "buffer overrun vulnerability" is due to sloppy C code. (And it takes a LOT of discipline to not be sloppy.)

C vs. C++

C++ is essentially a superset of C that supports type extension, object-oriented programming, and programming with generics.

The machine code generated for a body of C++ code is generally as fast and memory-efficient as the same code in C.

Like C, memory must be managed by hand, but various C++ facilities make memory management easier and safer than in C.

The additional power of C++ comes at a price: C++ is much more complicated than C and is packed with pitfalls and subtle complexities.

C vs. C++, continued

"C was invented so that people wouldn't have to write operating systems in assembler. C++ was invented so that people would haven't to write systems software in C".

—author unknown

In my opinion, there are only two situations where it is appropriate to choose C rather than C++ for a new system:

1. There is no usable C++ compiler and debugger for the target platform.

2. It is not practical and/or cost effective to train the development team in C++.

<u>But regardless of whether the destination is C or C++, the first step is to learn C.</u>

C vs. Java

In 1990 Sun Microsystems formed a group called the Green project. The initial focus was to create a software development environment for consumer electronics products.

C++ was the initial choice for a language for Green but frustration with C++ led to a new language, Oak.

Oak was renamed to Java.

Java borrows heavily from C++. Java's lexical rules, control structures, data types, and operators are very close to C.

Lots of what you've learned about Java is applicable in C.

When is C (or C++) a good choice?

Here are some scenarios that explore the question of choosing a language:

- A great amount of GNU Emacs is implemented in Lisp, which runs more slowly than C. Would it be worth the time rewrite the Lisp portions in C?

- Imagine a purchase decision between two small-office copiers. The primary consideration is cost but a close second is the copier's speed.

  The control software of one copier is written in C. The control software of the other is written in Java.

Which copier would be the better choice?

When is C (or C++) a good choice?, continued

Another:

- A company asks a consultant to write a program to perform a one-time migration of data for some number of users. The consultant knows C, C++, Java, Python, and Ruby. What are some factors that might be considered when choosing a language?

    - How many clients and how much data for each?
    - What's the likelihood this conversion will ever need to be done again?
    - How can the data be accessed?
    - Are there computations that require a particular library?
    - Must the final conversion be done in a fixed amount of time, like one evening?
    - Others?

iota.java vs. iota.c

iota.java vs. **iota.c**

```
public class iota {
  public static void main(String args[]) {
     int N = Integer.parseInt(args[0]);

    for (int i = 1; i <= N; i++)
      System.out.println(i);
  }
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *args[])
{
   int N = atoi(args[1]);

   for (int i = 1; i <= N; i++)
      printf("%d\n", i);
}
```

The C program above right is complete and compilable. By convention, C source files have the suffix **.c** (lower case). It is **fall15/c/iota.c**.

C functions can be distributed among files as the developer sees fit. There are no Java-like file-naming requirements. (More later on file organization.)

There are no classes in C—all executable code is held in functions.

Analogy: A C program is like a Java program with one class, named **Program**. Every executable line of code is in some method of **Program**.

iota.java vs. iota.c, continued

We'll be using the GNU C Compiler (gcc) to compile C programs.

<u>Put this alias in your .bashrc</u>: alias gcc="gcc -Wall -g -std=c1x" (Note: c-one-x.)
Slides and assignments will assume it is present. Once added, just do ***restart****.*

Let's use that gcc alias to create an executable file from iota.c.

```
% gcc iota.c

% ls -l a.out
-rwxrwxr-x 1 whm whm 9732 Sep 10 21:04 a.out

% file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=
0xa71dc06e817a7523c51b7c032409a65bfd713e27, not stripped
```

By default, the resulting executable is called a.out. Let's run it:

```
% a.out 3
1
2
3
```

iota.java vs. **iota.c**, continued

gcc's **-o** option can be used to specify a name for the executable:

```
% gcc -o iota iota.c

% ls -l iota
-rwxrwxr-x 1 whm whm 9732 Sep 10 21:58 iota

% iota 2
1
2
```

Unlike Java class files, which can be run by any (up to date) Java virtual machine on any platform, executables like **iota** are not portable between operating systems.

Let's try **iota** on my Mac:

```
% scp lec:cw/c/iota .

% iota
bash: iota: cannot execute binary file
```

iota.java vs. **iota.c**, continued

bash has a **time** builtin:

```
% time java mgrep ghi /usr/share/dict/words >/dev/null
real     0m0.357s  ("wall clock time")
user     0m0.240s  (CPU time in program code)
sys      0m0.012s  (CPU time in kernel code)
```

Let's predict...

```
% time java iota 3000000 >j
real     0m?.???s
user     0m?.???s
sys      0m?.???s

% time iota 3000000 >c
real     0m?.???s
user     0m?.???s
sys      0m?.???s

% diff c j

%
```

iota.java vs. **iota.c**, continued

Another area of contrast between Java and C is error handling:

```
% java iota
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
	at iota.main(iota.java:4)

% iota
Segmentation fault (core dumped)

% ls -l core
-rw------- 1 whm whm 237568 Sep 10 23:06 core
```

A **core** file is essentially a snapshot of the address space of the process.

Add **`ulimit -c 1000000`** to your **.bashrc** to enable core dumps.

The name "core" comes from the term "core dump", which in the early days was a word-by-word printed listing of memory contents used for debugging. A "core" is a tiny ferrite ring that magnetically stores one bit. Google for "core plane image".

iota.java vs. iota.c, continued

gdb is the GNU debugger. Given an executable and a core file generated when running it, **gdb** can be used to conduct a post-mortem analysis.

```
% gdb iota core
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
...more...
Reading symbols from /cs/www/classes/cs352/fall15/c/iota...done.
Program terminated with signal 11, Segmentation fault.
...more...
(gdb) where
#0  ____strtol_l_internal (nptr=0x0, endptr=0x0, base=10,
group=<optimized out>,
   loc=0x7f34501c1020) at ../stdlib/strtol_l.c:298
#1  0x00007f344fe42690 in atoi (nptr=<optimized out>) at atoi.c:28
#2  0x0000000000400566 in main (argc=1, args=0x7ffe103ddb88) at
iota.c:5
(gdb) frame 2
#2  0x0000000000400566 in main (argc=1, args=0x7ffe103ddb88) at
iota.c:5
5         int N = atoi(args[1]);
(gdb) p args[0]
$1 = 0x7ffe103de95a "iota"
```

Compilation of C programs

Syntax and semantics

The specification of a programming language has two key facets.

- Syntax:

  Specifies the sequences of symbols that are valid programs in the language.

- Semantics:

  Specifies the meaning of a sequence of symbols.

Here is an expression from **iota.java**:

```
Integer.parseInt(args[0])
```

Is the expression syntactically valid?

Yes.

How is validity determined?

The parser, one component of **javac**, is responsible for deciding that.

Syntax and semantics, continued

At hand:

    Integer.parseInt(args[0])

What's required for the expression above to be semantically valid?

- **args** must be an array.
- The subscript of **args** must be an integer.
- The class **Integer** must have a static public method named **parseInt** that accepts one argument, whose type must be compatible with the type of **args[0]**.

What's the type of **args[0]**?

    String

How can we see if **Integer** has a suitable **parseInt** method?

```
% javap java.lang.Integer | fgrep parseInt
 public static int parseInt(java.lang.String, int) throws ...
 public static int parseInt(java.lang.String) throws ...
```

Java compilation

Assignment 1 asked,

> *When a Java program is being compiled, what is the name of the file the Java compiler would consult to determine whether a call such as* **Rectangle r = new Rectangle(3, 4)** *is valid?*

Answer: **Rectangle.class**

At hand: **Integer.parseInt(args[0])**

javac will read **Integer.class** to see if **Integer** has a suitable **parseInt**.

Where is **Integer.class**?

```
% jar tvf /usr/local/jdk/jre/lib/rt.jar | fgrep java/lang/Integer.class
 8848 Wed Jun 05 20:49:42 MST 2013 java/lang/Integer.class
```

Integer.class is one of many files contained in the **rt.jar** *Java archive file*.

Where is **Integer.java**, which was compiled by **javac** to produce **Integer.class**?

```
% jar tvf /usr/local/jdk/src.zip | fgrep /Integer.java
46449 Wed Jun 05 19:01:38 MST 2013 java/lang/Integer.java
```

C compilation in slow motion

Let's check the size of **iota.c** and compile it again.

```
% wc  iota.c
  9  26 164 iota.c
% gcc -o iota iota.c
```

Here's a simplified version of what actually happens:

Step 1: The C *preprocessor* is run to "include" files, expand macros and more.

```
% gcc -E iota.c >iota-preprocessed.c

% wc iota.c iota-preprocessed.c
    9    26   164 iota.c
  938  2424 20188 iota-preprocessed.c
```

Step 2: The preprocessed C code is compiled into *assembly code* in a **.s** file.

```
% gcc -S iota-preprocessed.c

% wc iota-preprocessed.s
 359  743 5137 iota-preprocessed.s
```

C compilation in slow motion, continued

Step 3: The "Portable GNU assembler" (**as**) is run, producing an "object file" with suffix **.o**. (Despite the name, there is absolutely no relation between object-oriented programming and object files.)

```
% as --64 -o iota-preprocessed.o iota-preprocessed.s

% file iota-preprocessed.o
iota-preprocessed.o: ELF 64-bit LSB relocatable, x86-64, version 1
(SYSV), not stripped
```

nm can be used to show what *symbols* are present in an object file:

```
% nm iota-preprocessed.o
                 U atoi
0000000000000000 T main
                 U printf
```

The "**U**"s indicate **atoi** and **printf** are referenced but not defined in the **.o** file. The "**T**" indicates that **main** is defined. **main**'s code starts at offset **0**, shown as 64-bit hexadecimal number.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *args[])
{
   int N = atoi(args[1]);

   for (int i = 1; i <= N; i++)
      printf("%d\n", i);
}
```

C compilation in slow motion, continued

Step 4: The GNU "linker", **ld** ("LD") is used to create an *executable* from the object file. Conceptually, this is done,

```
% ld -o iota iota-preprocessed.o -lc
```

but the actual command is 866 characters long. (Do **gcc -v -o iota iota-preprocessed.o** to see it, and more.)

```
% file iota
iota: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=0x3f0347f91cda026d735ab975e5815e2219c969d9,
not stripped

% ls -l iota
-rwxrwxr-x 1 whm whm 9761 Sep 13 17:26 iota

% wc iota.c
  9  26 164 iota.c
```

C compilation in slow motion, continued

High-altitude view of the steps performed by **gcc -o iota iota.c**:

- Preprocess C source code to "include" files, expand macros and more.
- Compile the preprocessed code into assembly code (machine instructions).
- Assemble the assembly code into an object file.
- "Link" the object code and some library code into an executable.

Try **gcc**'s **-v** option, too: **gcc -v -o iota iota.c**

The above steps for compilation of C date back to the early days of C.

Important: **gcc** is an implementation of a C compiler. Another C compiler might do things entirely differently. The C11 standard doesn't dictate any particular organization for a C compiler.

In V7 UNIX the C compiler was launched with **cc**.

The C preprocessor

The idea of preprocessing

One of the cornerstones of C is the idea of preprocessing.

The C preprocessor reads C source code and performs simple textual substitutions based on *preprocessor directives*, which are indicated by lines that start with #.

iota.c starts with two preprocessor directives:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *args[])
...
```

The line #include <stdio.h> directs the preprocessor to temporarily suspend reading lines from the current source file and instead read lines from the standard *header file* named stdio.h. When all lines have been read from stdio.h, processing of the original source file continues.

In effect, #include causes the text of stdio.h to be inserted in the current stream of source code, just as if it had been pasted in with an editor.

More on **#include**

The use of angle-brackets in **#include <stdio.h>** indicate that **stdio.h** is a well-known file that's found in an implementation-defined directory, which on most systems is **/usr/include**.

Let's see if **stdio.h** has any **#include** directives:

```
% fgrep "#include" /usr/include/stdio.h
#include <libio.h>
#include <bits/stdio_lim.h>
#include <bits/sys_errlist.h>
```

gcc's **-E** option sends preprocessor output to standard output. Let's see how many lines are generated by preprocessing **iota.c**:

```
% gcc -E iota.c | wc
   938   2424   20188
```

Preprocessing turns **iota.c**'s 9 lines into 938 lines!

Quick excursion: Let's try **fall15/c/iota.cc**, a version of **iota.c** in C++:

```
% g++ -E iota.cc | wc
 19749   43480  459002
```

Why are we preprocessing?

Let's comment-out those **#include**s, to see if we really need them.

```
% cat iota.c
//#include <stdio.h>
//#include <stdlib.h>
int main(int argc, char *args[])
{
   int N = atoi(args[1]);

   for (int i = 1; i <= N; i++)
      printf("%d\n", i);
}

% gcc iota.c
iota.c: In function ‘main’:
iota.c:5:5: warning: implicit declaration of function ‘atoi’…
iota.c:8:9: warning: implicit declaration of function ‘printf’…
iota.c:8:9: warning: incompatible implicit declaration of built-in
function ‘printf’ [enabled by default]
```

Why are we preprocessing, continued?

With no **#include**s, we get some warnings:

```
% gcc iota.c
iota.c: In function ‘main’:
iota.c:5:5: warning: implicit declaration of function ‘atoi’…
iota.c:8:9: warning: implicit declaration of function ‘printf’…
...
```

Java uses **.class** files to check semantics (like **Integer.parseInt(args[0])**).

<u>In C we use **#include**s of header files to specify function signatures for the compiler to use to check semantics.</u>

Let's reinstate the **#include**s and see where **atoi** and **printf** appear in the preprocessor output (which is the compiler input!)

```
% gcc -E iota.c | egrep -w "atoi|printf"
extern int printf (__const char *__restrict __format, ...);
extern int atoi (__const char *__nptr)
   int N = atoi(args[1]);
      printf("%d\n", i);
```

Why are we preprocessing, continued?

At hand:

```
% gcc -E iota.c | egrep -w "atoi|printf"
extern int printf (__const char *__restrict __format, ...);
extern int atoi (__const char *__nptr)
  int N = atoi(args[1]);
    printf("%d\n", i);
```

We'll later understand those *prototypes* for **printf** and **atoi** but for now we'll think of them this way:

- **printf** takes one or more arguments, the first of which is a string.
- **atoi** takes a single argument, a string.
- Both return an integer value.

TL;DR: One use for **#include**s is to specify function prototypes that the compiler can use to check the semantics of function calls.

Note that we might need to include a file with thousands of prototypes just so the C compiler can check a single function call!

Minimums and maximums

This program shows the range of an **int** in Java:

```
public class intminmax { // in fall15/c
   public static void main(String args[]) {
      System.out.format("int range is %d to %d\n",
          Integer.MIN_VALUE, Integer.MAX_VALUE);
   }
}
```

Here is its counterpart in C:

```
#include <stdio.h>
#include <limits.h>
int main(int argc, char *args[])
{
   printf("int range: %d to %d\n", INT_MIN, INT_MAX);
}
```

What's new?

The **#define** preprocessor directive

At hand:

```
#include <stdio.h>
#include <limits.h>
int main(int argc, char *args[])
{
   printf("int range: %d to %d\n", INT_MIN, INT_MAX);
}
```

Let's look in **limits.h**:

```
% grep INT_M /usr/include/limits.h
#  define INT_MIN       (-INT_MAX - 1)
#  define INT_MAX       2147483647
#  define UINT_MAX      4294967295U
```

The first **#define** above specifies that the preprocessor is to replace all occurrences of **INT_MIN** with **(-INT_MAX - 1)**.

We also see replacements for **INT_MAX** and **UINT_MAX** (the largest *unsigned* **int** value.)

#define, continued

```
# define INT_MIN      (-INT_MAX - 1)
# define INT_MAX       2147483647
```

At hand:

```
#include <stdio.h>
#include <limits.h>
int main(int argc, char *args[])
{
   printf("int range: %d to %d\n", INT_MIN, INT_MAX);
}
```

Let's see what the preprocessor does with **intminmax.c**. Note the **tail -7**.

```
% gcc -E intminmax.c | tail -7
# 8 "/usr/lib/gcc/x86_64-linux-gnu/4.6/include-fixed/syslimits.h" 2 3 4
# 35 "/usr/lib/gcc/x86_64-linux-gnu/4.6/include-fixed/limits.h" 2 3 4
# 3 "intminmax.c" 2
int main(int argc, char *args[])
{
   printf("int range: %d to %d\n", (-2147483647 - 1), 2147483647);
}
```

INT_MIN and **INT_MAX** are said to be *macros*—they expand into text that replaces an identifier.

An **iprint** macro

Macros can have parameters. Here's a macro that will help us explore C:

```
% cat /cs/www/classes/cs352/fall15/h/iprint.h
#define iprint(e) printf(#e " = %d\n", e)
```

See slide 40!

Let's extend our **gcc** alias so we can use **#include "iprint.h"**. Note the quotes.

```
alias gcc="gcc -Wall -g -std=c1x -I/cs/www/classes/cs352/fall15/h"
```

The **-I** says "Look in this directory for header files, too."

Here's a program that uses it: (**fall15/c/iprint1.c**)

```
#include <stdio.h>
#include "iprint.h"
int main(int argc, char *args[])
{
    iprint(5 + 7 * 3);
    iprint('A' + 25);
    iprint(argc);
}
```

Execution:

```
% gcc iprint1.c
% a.out
5 + 7 * 3 = 26
'A' + 25 = 90
argc = 1
```

Can we create a Java analog for **iprint**?

There's lots more with the preprocessor

We've seen the preprocessor used for including files and macro replacement. It does more, including *conditional compilation*.

Along with function prototypes and macros, header files contains ***struct**s*, ***typedef**s*, ***const*** values and other things that we'll be learning about.

There are lots of header files on lectura:

```
% find /usr/include -name \*.h | wc -l
8869

% find /usr/include -name \*.h -exec cat {} \; | wc -l
1492328
```

The preprocessor is very powerful but has many pitfalls, too. We'll learn all about it soon but we know what we need to know for now.

Sidebar: How the **iprint** macro works

Here's **iprint**:

```
#define iprint(e) printf(#e " = %d\n", e)
```

Let's use it:

```
iprint(3 + 4);
```

Here's what that "call" of **iprint** turns into:

```
printf("3 + 4" " = %d\n", 3 + 4);
```

We can see that the text **3 + 4** has replaced the **e** at the end of the expansion but what's going on with **#e**?

Rule 1: The form **#*param*** says to insert **param** but wrap it in quotes, making it a string literal.

Rule 2: Juxtaposed literals are considered to be a single string literal.

The end result is that **iprint(3 + 4);** is equivalent to this:

```
printf("3 + 4 = %d\n", 3 + 4);
```

Bits

What's a bit?

Wikipedia says,

"A bit is the basic unit of information in computing and digital communications. A bit can have only one of two values, and may therefore be physically implemented with a two-state device.

...

"The term bit is a *portmanteau* of binary digit."

With N bits we can represent 2^N unique values:

Number of bits	Number of unique values
1	2
2	4
8	256
16	65,536
32	4,294,967,296
64	18,446,744,073,709,551,616

How much data can three bits hold?

Don't say it but think of a number: What's the largest number three bits can hold?

Here is a simple representation

Bits	Integer
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Here is a "two's complement" representation

Bits	Integer
100	-4
101	-3
110	-2
111	-1
000	0
001	1
010	2
011	3

Here's another representation

Bits	Integer
000	0
001	one million
010	one billion
011	one trillion
100	one quadrillion
101	one quintillion
110	one googol
111	Aleph-null, the first transfinite cardinal number

Bits and computer architecture

Computers are often characterized as being N-bit machines. Examples:

Machine	Introduced	Bits
IBM 709	1958	36
IBM 360	1964	32
PDP-7	1965	18
PDP-11	1970	16
Intel 4004	1971	4
Intel 8080	1974	8
Intel 8086	1978	16
VAX-11	1978	32
Intel 80386	1985	32
Intel Pentium	1993	32
Pentium with x86-64	2003	64

Vast simplification: An N-bit machine works efficiently with "words" of data that comprise N bits.

Conceivably, a 32-bit machine might take longer to store 16 bits than 32 bits.

Basic types in C

The *basic types*

Section 6.2.5 of the C11 standard defines three groups of *basic types*:

- **char**

- Signed and unsigned integer types
  Examples: **int**, **short**, **long**, **long long**, **unsigned int**

- Floating types
  Examples: **float**, **double**, **long double**, **double _Complex**

The basic types are *scalar* types—they hold a single value.

Reminder: A good-enough copy of the C11 standard can be found in **fall15/ c11.pdf**

Sidebar: integral types in Java

In Java, the range of integral types is defined by The Java Language Specification.

Here is the full text of of 4.2.1 from **fall15/jls8.pdf**:

4.2.1 Integral Types and Values

The values of the integral types are integers in the following ranges:

- For `byte`, from -128 to 127, inclusive
- For `short`, from -32768 to 32767, inclusive
- For `int`, from -2147483648 to 2147483647, inclusive
- For `long`, from -9223372036854775808 to 9223372036854775807, inclusive
- For `char`, from `'\u0000'` to `'\uffff'` inclusive, that is, from 0 to 65535

Note that nothing is mentioned about sizes in bits!

But it is common for JVM implementations to use 8 bits for **byte**s, 16 bits for **short**s, etc.

Ranges for types in C

In C, the range of values for each scalar type is "implementation dependent", but minimum magnitudes (positive and negative) are specified. A sampling:

short	-32767 to +32767
int	-32767 to +32767
long	-2147483647 to +2147483647
long long	-9223372036854775807 to +9223372036854775807
char	-127 to 127 if **char**s are signed; 0 to 255 if not

CSC 252 alumni: What's a little surprising about those ranges?

There are minimums for the floating types, too. The C11 standard has several pages of specifications but a simplified example is that a **double** must be able to represent numbers in the range 10^{-37} to 10^{37} and maintain 10 significant digits.

At one point in time it was typical for an N-bit machine to have N-bit **int**s, but that's no longer as common as it was.

Fixed vs. varying ranges

Java specifies fixed ranges for integral types but C lets sizes vary across implementations, albeit with minimums.

What's a good thing about Java's fixed ranges?

The programmer doesn't need to make provisions for something like an unusually small **int**. If an **int** calculation "fits" on JVM, it will fit on all JVMs.

What's a good thing about implementation-defined ranges?

A 16-bit processor in an embedded system isn't burdened with manipulating 32-bit **int** values.

C has gracefully handled forty years of memory growth. Java is already in a pinch with Unicode 2.0, which supports more than 65,535 characters.

The C11 standard requires **<limits.h>** to define *TYPE*_MIN and *TYPE*_MAX macros for all types. The programmer can use the preprocessor to make a decision at compile time to use **long**s instead of **int**s, should **int**s be too small.

The **sizeof** operator

The **sizeof** operator looks like a function. We can use it to see how much memory is occupied by an instance of a type.

```
#include <stdio.h>
int main(int argc, char *args[]) // sizeof0.c
{
   printf("sizeof(char) is %lu\n", sizeof(char));
   printf("sizeof(int) is %lu\n", sizeof(int));
}
```

sizeof produces a result of type **unsigned long**. Oops: **%zu** is better!

Execution:

```
% a.out
sizeof(char) is 1
sizeof(int) is 4
```

The standard says **sizeof(char)** is always 1. **char**s are almost always one byte in size, so we'll think of **sizeof(...)** as producing a size in bytes.

We can see that **int**s are apparently four bytes long. Assuming 8-bit bytes, we're working with 32-bit **int**s. But, **INT_MAX** tells us what an **int** can hold!

sizeof, continued

Let's look at the sizes of various types. We'll use a **psize** macro to avoid repetition.

```
#include <stdio.h>
#define psize(e) printf("sizeof(" #e ") = %lu\n", sizeof(e))

int main(int argc, char *args[]) // sizeof1.c
{
    psize(char);
    psize(short);
    psize(int);
    psize(long);
    psize(sizeof(sizeof(int)));
    psize(float);
    psize(double);
    psize(long double);
    psize(long double _Complex);
}
```

Execution:

```
% a.out
sizeof(char) = 1
sizeof(short) = 2
sizeof(int) = 4
sizeof(long) = 8
sizeof(sizeof(sizeof(int))) = 8
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 16
sizeof(long double _Complex) = 32
```

Just like **iprint(e)**, **psize** is writing code for us! Try this: write a program that generates the above output but don't use a macro.

sizeof, continued

Prove this: The value of a **sizeof** expression is computed at compile time.

Proof:

```
% cat sizeof1a.c
#include <stdio.h>
int main(int argc, char *args[])
{
    printf("%d %lu %lu %lu %d\n",
        111, sizeof(double),
        sizeof(long double),
        sizeof(long double _Complex),
        999);
}

% gcc -S sizeof1a.c
```

Why did we put in 111 and 999?

So we can look for them!

Excerpts from **sizeof1a.c**

```
    ...
.LC0:
    .string "%d %lu %lu %lu %d\n"
    ...
main:
    ...
    movl   $.LC0, %eax
    movl   $999, %r9d
    movl   $32, %r8d
    movl   $16, %ecx
    movl   $8, %edx
    movl   $111, %esi
    movq    %rax, %rdi
    movl   $0, %eax
    call   printf
    movl   $0, %eax
    ...
```

Variables and constants

Variables

As in Java, scalar variables are declared by specifying a type followed by one or more identifiers:

```
char c;
int min, max;
float minSkew, fudgeFactor;
long initial_count, last_count;
```

The standard specifies that identifiers are composed of letters, digits, and underscores, and may not start with a digit.

Some compilers, including gcc, also permit a dollar sign.

As in Java, identifiers are case sensitive.

Most compilers, including gcc, permit very long identifiers but the C11 standard guarantees only 31 significant characters in some cases. (The C90 standard guarantees only 6 in some cases.)

Constants

Lexical tokens such as **20**, **'x'**, and **12.34** are called *literals* in Java. In C they are called *constants*.

As in Java, integer constants can be specified in decimal, hexadecimal (base 16), or octal (base 8):

```
int ten = 10;
int twenty = 0x14;      // 0x indicates hexadecimal
int thirty = 036;       // leading zero indicates octal
```

By default, the type of an integer constant is **int**.

The **L** and **LL** suffixes can be added to explicitly indicate a **long** or **long long** constant:

```
long two_billion = 2000000000L;
long long three_trillion = 3000000000000LL;
```

Constants, continued

C code is said to be "portable" if it will operate correctly with any C compiler that meets the standard, regardless of the underlying machine architecture.

Is the following C code portable? Why or why not?

```
int one_billion = 1000000000;
```

No. On a machine with 16-bit `int`s, for example, the value is too big for an `int`!

If in Java instead, is the code portable?

Yes. The JLS guarantees that `1000000000` is in the range of an `int`.

Constants, continued

Floating point constants have a familiar form:

```
float a = 123.456;
double b = -1e3;
double c = 1234e-10;
```

By default, the type of a floating point constant is **double**.

A suffix can be added to a floating point constant to force a type:

```
111.222F
333.444D
555.666L  (long double)
```

Constants, continued

Let's use the **psize** macro (now in **fall15/h/352.h**) to explore sizes of constants.

```
% cat constants1.c
#include <stdio.h>
#include "352.h"

int main(int argc, char *args[])
{
    psize(3);
    psize(3LL);
    psize(3.0);
    psize(3.0F);
    psize(3.0L);
}
```

Execution:

```
% gcc constants1.c && a.out
sizeof(3) = 4
sizeof(3LL) = 8
sizeof(3.0) = 8
sizeof(3.0F) = 4
sizeof(3.0L) = 16
```

Be sure you've got the latest **gcc** alias:

```
alias gcc='gcc -Wall -g -std=c1x -I/cs/www/classes/cs352/fall15/h'
```

Character constants

Character constants are specified by enclosing a character in single quotes.

```
char let_a = 'a', dollar = '$';
```

In C, a character constant is simply another way to specify an **int** constant.

Consider this code:

```
psize('a');
printf("'a' = %d, '$' = %d\n", 'a', '$');
```

Output:

```
sizeof('a') = 4
'a' = 97, '$' = 36
```

Character constants, continued

A **char** can be initialized with an ordinary integer constant.

```
char c1 = 42, c2 = 50, c3 = 65, c4 = 98;
printf("c1 = %c, c2 = %c, c3 = %c, c4 = %c\n", c1, c2, c3, c4);
```

The **%c** format specifier causes **printf** to produce the character whose ASCII code is specified by the value of the corresponding argument.

Output:

```
c1 = *, c2 = 2, c3 = A, c4 = b
 //  42     50     65     98
```

It is important to understand that **char** values are simply small integers. Programs such as ssh clients and text editors display those integer values in a familiar graphical form. Above we see that **42** is shown as *****, **50** is shown as **2**, etc.

Let's use **echo** and **od** to explore character values:

```
% echo "*2Ab <=> GHI" | od -Ad -td1
0000000  42  50  65  98  32  60  61  62  32  71  72  73  10
```

Sidebar: ASCII

"ASCII" stands for American Standard Code for Information Interchange.

The ASCII standard assigns a number from 0-127 to 128 specific characters.
Here's one view of the "encoding":

0	NUL
1-26	^A through ^Z but many have other names. ^J (10) is linefeed–LF (or newline); ^M (13) is carriage return–CR.
27-31	Various control codes including ESC (27)
32-47	space ! " # $ % & ' ( ) * + , - . /
48-57	The digits 0-9
58-64	: ; < = > ? @
65-90	A-Z
91-96	[\ ] ^ _ `
97-122	a-z
123-127	{ | } ~ DEL (another control code)

There are many ASCII charts on the web but **man ascii** is very convenient!

ASCII, continued

Why do we need a standard like ASCII?

Because computers store numbers, not characters!

When we type "Hello" into an editor and hit Save, five numbers get stored in a file on the disk. What five numbers should we store?

If we encode those five letters using ASCII, we store `72 101 108 108 111`.

If we encode with them with EBCDIC, we store `200 133 147 147 150`.

If an editor reads a file and gets `101 993 433 32 48 49 726`, what characters should it display to the user?

If we don't know what encoding was used, we have no idea what characters those numbers represent.

Could it be ASCII?

It is said that at the time ASCII was created, over 60 character encoding systems were in use. (Google for Bob Bemer.)

ASCII, continued

What are the two fundamental decisions when designing a character encoding system?

- The characters in the set.
- The ordering of the characters.

What's the fundamental trade-off when deciding what characters to include in an encoding system?

The larger the set, the more bits it takes to store a single character.

Consider some sizes:

- ASCII is a 7-bit encoding—128 characters
- EBCDIC is an 8-bit encoding—256 characters
- RADIX-50 has 40 characters: `0-9 A-Z . $ %` and space
- Unicode was originally a 16-bit code but the latest version defines 1,114,112 "code points".

Impress your friends by memorizing the ASCII encoding!

http://memrise.com/course/80243/ascii-to-decimal/

(Back to) Character constants

Just as in Java, various character "escapes" are available:

```
#include <stdio.h>
int main(int argc, char *args[])
{
    char quote = '\'', newline = '\n', bslash = '\\';

    char lsquare = '\x5b', rsquare = '\135';

    printf("%c%c%c%c%c",
        lsquare, quote, newline, bslash, rsquare);
}
```

Execution:

```
% PS1="% "
% gcc constants2.c
% a.out
['
\]% a.out | wc -c
5
%
```

```
% a.out | od -td1        (decimal)
0000000   91   39   10   92   93

% a.out | od -tx1        (hexadecimal)
0000000 5b 27 0a 5c 5d

% a.out | od -to1        (octal)
0000000 133 047 012 134 135
```

String literals

In C, a sequence of characters enclosed in double quotes is called a *string literal*.

As we saw in **iota.c**, a string literal is a suitable first argument for **printf**:

```
printf("%d\n", i);
```

"%d\n" looks just like a Java string literal, which has the type **String**.

However, the meaning of "%d\n" is very different in C:

1. It causes creation of a nameless **char** array with four values: 37, 100, 10, and 0.

2. The value of "%d\n" is the address of the first value in the array.

We'll learn all about string literals later. For now we'll use string literals only as **printf** format specifiers.

Initialization of variables

Java has a "definite assignment" rule that requires that a local variable be initialized or assigned a value before it is used. If it can't be proven that some value has definitely been assigned to a local variable, it is an error to use it.

There is no similar requirement in C. Example:

```
% cat init.c
...
int main(int argc, char *args[])
{
   double x, y; int i, j;
   printf("i + j = %d, x + y = %g\n", i + j, x + y);
}
% gcc init.c
...warnings...
% a.out
i + j = -4128880, x + y = -7.67023e+304
```

The values of **i**, **j**, **x**, and **y** are simply the values that happen to be in the memory locations associated with those variables. Your results may vary.

Initialization of variables, continued

At hand:

- Java considers it to be an error if a local variable might be used before it is given a value.

- C does not attempt to verify that a local variable has been given a value before it is used.

Question: What are the implications of this C design choice in...

Language specification?

Simpler

Compilation?

Faster (and a simpler, smaller compiler)

Execution?

Faster—no need to "zero" memory for local variables—but carries a risk.

The **-Wall** option in our **gcc** alias causes warnings to be produced for **init.c**.
With **\gcc init.c**, which directs bash to ignore the alias, there are no warnings!

lint(1) was used to look for suspicious code years ago. Check Wikipedia!

Operators

Multiplicative operators

C has three multiplicative operators:

`*   /   %`

In broad terms, the operators match their Java counterparts, performing multiplication, division, and remaindering.

The operands of % must be integers.

As in Java, if one operand of * or / is integral and the other is floating point, the result is floating point.

Examples:

```
5 * 7           //  35
5.5 * 7.7       //  42.35
5 * 7.7         //  38.5
34 / 3          //  11
34.0 / 3        //  11.3333
34 % 3          //  1
```

Multiplicative operators, continued

With integer division in C90, if either operand of `/` is negative, whether the quotient rounds towards zero or away from zero is implementation specific. For example, `-13/4` might produce `-3` or `-4`.

In C99 and C11, the quotient of two integers is "the algebraic quotient with any fractional part discarded"—it "truncates toward zero". (C11 6.5.5)

Example:

```
5 / -3          // -1
-5 / -3         // 1
-5 / 3          // -1
5.0 / -3.0      // -1.66667
-5.0 / -3.0     // 1.66667
-5.0 / 3.0      // -1.66667
```

The C90, C99, and C11 standards have the same specification regarding division or remaindering by zero:

> *"In both operations, if the value of the second operand is zero, the behavior is undefined."*

The phrase "the behavior is undefined" indicates that for this situation, an implementation can do whatever it wants (possibly whatever is easiest to implement) and still be compliant with the standard.

Multiplicative operators, continued

With **gcc** on lectura, a floating point division by zero produces a representation of infinity, but an integer division by zero produces a core dump:

```
% cat div2.c(Note: #include not shown...)
int main(int argc, char *args[])
{
   double q = 1.0 / 0.0;          // No warning produced
   printf("q = %g\n", q);

   int q2 = 1 / 0;
   printf("q2 = %d\n", q2);    // Produces a warning
}

% gcc div2.c
div2.c: In function 'main':
div2.c:8:16: warning: division by zero [-Wdiv-by-zero]

% a.out
q = inf
Floating point exception (core dumped)
```

Problem: Somebody says, "The **printf** is blowing up in the middle of printing **q = inf** —it should be **q = infinity**, like in Java" How could we address that claim?

+ and -

C has two binary arithmetic additive operators:

```
+   -
```

C has two unary arithmetic operators:

```
+   -
```

All four perform just like their Java counterparts.

Examples:

```
4 + 7           // 11

1e4 + 1e7       // 1.001e+07

1.2 - 3         // -1.8

-(3 - 4)        //  1

+(3 - 4)        // -1
```

The assignment operator

C's assignment operator is essentially identical to Java's.

Assuming that **i** is declared as **int**, evaluating the expression **i = 7** does two things:

1. It produces the value **7** as the result of the expression.

2. As a *side-effect*, the value held in **i** is changed to **7**.

Stated more briefly:

The assignment operator produces a value but has a side effect, too.

Example:

```
int i = 3;
iprint(i = 7);
    // Output: i = 7 = 7
iprint(i);
    // Output: i = 7
```

(demonstrates that **i = 7** *produces* **7***)*

(demonstrates that **i** *was changed to* **7***)*

Sidebar: Side effects

A "side effect" is a change in the state of the program's observable data or in the state of the environment with which the program interacts.

Which of these Java expressions have a side effect?

```
x + 3 * y
```

No side effect. A computation was done but no evidence of it remains.

```
s.length() > 2 || s.charAt(1) == '#'
```

No side effect. A computation was done but no evidence of it remains.

```
"testing".toUpperCase()
```

A string ***"TESTING"*** *was created somewhere but we can't get to it.*

L.add("x"), where **L** is an **ArrayList**

Side effect: An element was added to ***L****. Definitely a side-effect!*

```
System.out.println("Hello!")
```

Side effect: ***"Hello!"*** *went somewhere.*

```
window.checkSize()
```

Can't tell without looking at ***window.checkSize()****!*

The assignment operator, continued

Given `int i, j = 3, k;`, what does the following statement do?

```
i = j + (k = 2);
```

1. `k = 2` is evaluated first, producing the value `2`. `k` will be set to `2` as a side-effect, but exactly when that happens is somewhat indefinite.
2. `j + 2` is evaluated next, producing `5`.
3. `i = 5` is evaluated next, producing `5`. `i` will be set to `5` as a side-effect.
4. It is guaranteed that `i` will be set to `5` and `k` will be set to to 2 before execution of the next statement begins.

The assignment operator, continued

Another example:

```
% cat asgnop1.c
#include <stdio.h>
int main(int argc, char *args[])
{
   char c;  int i;

   printf("c = %c, i = %d, c = %c\n", c = 'A', i = 10, c = 'B');
   printf("c = %c\n", c);
}

% gcc asgnop1.c
asgnop1.c: In function ‘main’:
asgnop1.c:7:59: warning: operation on ‘c’ may be undefined
      [-Wsequence-point]

% a.out
c = A, i = 10, c = B
c = A
```

Because the ordering of the assignments to **c** is undefined, we get this warning!

The standard defines a number "sequence points"—points at which it is guaranteed that any side-effects have been completed. One such point is between statements.

Assignment vs. initialization

It's important to distinguish assignment from initialization.

This is an assignment statement:

```
i = 7;
```

This is a declaration with initialization:

```
int i = 7;
```

Is the following line a declaration, an assignment, or both?

```
double x, y = 0.0;
```

x is declared.
y is declared and initialized.
There is no assignment.

Compound assignment operators

C has *compound assignment* for the multiplicative and additive binary operators:

```
+=   -=   *=   /=   %=
```

In essence, they perform like their Java counterparts. Generally stated,

E1 op= E2

is equivalent to

E1 = E1 op E2

with the exception that ***E1*** is evaluated only once.

Examples, assuming `int i = 5;`

```
i *= 3;   // assigns 15 to i

i %= 2; // assigns 1 to i
```

Increment and decrement operators

C has prefix and postfix increment and decrement operators that behave like their Java counterparts.

Sometimes they are simply used for their side-effect:

```
int i = 5;

i++;
// i is now 6

i--;
// i is 5

--i;
// i is 4

++i;
// i is 5
```

But along with having a side-effect, they each produce a value, too!

Increment and decrement, continued

It is critical to understand that the **++** and **--** operators produce a value AND, as a side-effect, alter the value of the associated variable.

What is the value of the following expression?

```
i++
```

The value of **i++** is **i**. As a side effect, **i** is incremented.

Example:

```
int i = 5;

iprint(i);
    // Output: i = 5
iprint(i++);
    // Output: i++ = 5
iprint(i);
    // Output: i = 6
iprint(i++);
    // Output: i++ = 6
```

Increment and decrement, continued

At hand: The **++** and **--** operators produce a value AND, as a side-effect, alter the value of the associated variable.

What is the value of the following expression?

```
i--
```

The value of **i--** is **i**. As a side effect, **i** is decremented.

Example:

```
int i = 5;

iprint(i);
     // Output: i = 5
iprint(i--);
     // Output: i-- = 5
iprint(i);
     // Output: i = 4
iprint(i--);
     // Output: i-- = 4
```

Increment and decrement, continued

The meaning of the prefix forms are different, but just like Java:

The value of ++i is i + 1. As a side effect, i is incremented.

The value of --i is i - 1. As a side effect, i is decremented.

Example:

```
int i = 5;

iprint(i++);
    // Output: i++ = 5
iprint(i);
    // Output: i = 6
iprint(++i);
    // Output: ++i = 7
iprint(i);
    // Output: i = 7
```

Sidebar: Value, type, and side-effect

An *expression* is a sequence of symbols that can be evaluated to produce a value.

Here are some Java expressions:

```
'x'
i + j * k
f(args.length * 2) + n
```

There are three questions that programmers often consider when looking at an expression in conventional languages like C and Java:

- What value does the expression produce?
- What's the type of that value?
- Does the expression have any side effects?

Mnemonic aid: Imagine you're wearing a vest that's reversed. "vest" reversed is "t-se-v": type/side-effect/value.

Sidebar: Value, type, and side-effect, continued

What is the value of the following Java expressions?

```
3 + 4
    7

1 < 2
    true

"abc".charAt(1)
    'b'

s = "3" + 4
    "34"

"a,bb,c3".split(",")
```
An array with three elements: "a", "bb" and "c3"

```
"a,bb,c3".split(",")[2]
    "c3"

"a,bb,c3".split(",")[2].charAt(0) == 'X'
    false
```

Sidebar: Value, type, and side-effect, continued

What is the type of the value produced by each of the following Java expressions?

```
3 + 4
     int

1 < 2
     boolean

"abc".charAt(1)
     char

s = "3" + 4
     String

"a,bb,c3".split(",")
     String []

"a,bb,c3".split(",")[2]
     String

"a,bb,c3".split(",")[2].charAt(0) == 'X'
     boolean
```

When we ask,
"What's the type of this expression?"

we're actually asking this:
"What's the type of the value produced by this expression?"

Sidebar: Value, type, and side-effect, continued

What are the side effects of each of the following Java expressions?

3 + 4

No side effects

1 < 2

No side effects

"abc".charAt(1)

No side effects

s = "3" + 4

s will now reference a **String** with value "**34**".

What if **s** was already "**34**"? Is it still considered a side-effect?

"a,bb,c3".split(",")

No side effects

"a,bb,c3".split(",")[2]

No side effects

"a,bb,c3".split(",")[2].charAt(0) == 'X'

No side effects

Sidebar: Value, type, and side-effect, continued

Here's a Java program that makes use of the "autoboxing" mechanism to show the type of values. (Note that if there were a standard Java preprocessor we wouldn't need to use `n++` to associate "inputs" with output.)

```
public class exprtype {
    public static void main(String args[]) {
        int n = 1;
        showtype(n++, 3 + 'a');
        showtype(n++, 3 + 4.0);
        showtype(n++, "a,b,c".split(","));
        showtype(n++, new HashMap<String,Integer>());
    }
    private static void showtype(int num, Object o) {
        System.out.format("%d: %s\n", num, o.getClass());
    }
}
```

Output:

```
1: class java.lang.Integer
2: class java.lang.Double
3: class [Ljava.lang.String;
4: class java.util.HashMap
```

(Note: no String or Integer—type erasure!)

Relational and equality operators

C provides the same set of relational and equality operators that Java does.

```
< > <= >= != ==
```

The C operators perform the same tests as their Java counterparts, but the return type is different:

In C, relational and equality operators produce a result of type **int**. If the comparison succeeds, the result is 1. Otherwise it is 0.

Examples:

```
3 <= 4
    // produces 1

'a' == 'b'
    // produces 0

5.0 > 4.9
    // produces 1

sizeof(8 >= 9)
    // produces sizeof(int)
```

Relational and equality operators, continued

Does each of the following expressions compile? If so, what is its value?

```
1 < 2 + 3 == 4

x == y != 'z'

1 < 2 < 3

1 < 2 < 3 > 4 > 5

1 < 2 < / > 4 > 5

1 < 2 / 4 > 5
```

Try them with **iprint**! Then write six more yourself.

If you post one on Piazza that makes me say, "Interesting!", I'll give you an assignment point.

Logical operators

C has the same three logical operators as Java:

```
|| && ! (unary)
```

The operands of the logical OR operator (**||**) must be **int**s or convertible to **int**s.

Logical OR produces an **int** result of 1 if either of its operands has a non-zero value. Otherwise, 0 is produced.

Examples:

```
1 || 0
    // produces 1

2 > 3 || 4 != 5
    // produces 1
```

Two more:

```
x == x + 1 || y == y - 2

int result = x < y || y > z * 5;
```

Logical operators, continued

As in Java, *short-circuit evaluation* is used:

The right-hand operand of a logical OR is evaluated only if the left-hand operand yields zero.

Example:

```
int i = 0;

3 < 4 || i++  // produces 1, i is unchanged

3 > 4 || i++  // produces 0, i is incremented
```

Strong recomendation: Work through the code above!

Logical operators, continued

The logical AND operator (**&&**) produces 1 (an **int**) if both operands have a non-zero value. Otherwise, it produces 0.

Examples:

```
1 && 0
    // produces 0

1 < 2 && 3 != 4
    // produces 1

('a' || 'b') && 'c'
    // produces 1
```

Short-circuit evaluation is used: The right-hand operand is evaluated only if the left-hand operand yields a non-zero result.

Logical operators, continued

The logical NOT operator (!) produces an **int** result of 1 if its operand is zero. It produces 0 otherwise.

```
!0
    // produces 1

!1
    // produces 0

!2
    // produces 0

!!-2
    // produces 1

!(1 < 2 && 0)
    // produces 1
```

The conditional operator

The *conditional operator* in C looks like its counterpart in Java:

```
E1 ? E2 : E3
```

First, the expression `E1` is evaluated. If it produces a non-zero value, `E2` is evaluated and its result becomes the result of the conditional expression.

Alternatively, if `E1` produces a zero value, `E3` is evaluated and its result becomes the result of the conditional expression.

Examples:

```
1 ? 2 : 3
    // produces 2

1 < 2 ? 3 < 4 : 5
    // produces 1
```

The conditional operator, continued

What does the following call do?

```
printf("Process complete; %d box%s  filled.\n",  n, n > 1 ? "es" : "");
```

A simple view for now: the `%s` format specifier indicates that the corresponding argument is a string literal.

Here are two potential results:

```
Process complete; 1 box filled.

Process complete; 3 boxes filled.
```

FYI, Java analog:

```
System.out.format("Process complete; %d box%s filled.\n",
    n, n > 1 ? "es":"");
```

C11's boolean type

C11 defines a boolean type: **_Bool**

The **<stdbool.h>** header has macros for **bool**, **true**, and **false**. Example:

```
% cat bool.c
#include <stdbool.h>
int main(int argc, char *args[])
{
   bool flag;
   bool a = true;
   bool b = false;
}

% gcc -E bool.c | tail -5
{
   _Bool flag;
   _Bool a = 1;
   _Bool b = 0;
}
```

My opinion is that using C11's booleans does little more than add a layer of obscurity. You're free to use them, but I'll probably continue to avoid them.

Precedence and associativity

In essence, the precedence and associativity of C operators matches their Java counterparts.

You might think the C11 standard would have a simple chart that shows operator precedence and associativity but if there is one, I can't find it!

Neither does JLS8 have a precedence chart!

Why not?

I don't know!

The C11 standard and the JLS define a *grammar* for their respective languages, and precedence is implied by that grammar. My guess is that authors felt it would be redundant to include a simple chart. That seems pretty silly!

This seems like a classic example of "works in theory but not in practice".

fall15/cpred.png is a shot of the precedence chart from section 2.12 in K&R2e. There's a link on the Piazza resources page, too.

A trio of control structures

Expressions vs. statements

Just like Java, C makes a distinction between expressions and statements.

Here are three expressions:

```
x = 7
i++
printf("Hello\n")
```

Appending a semicolon to an expression makes it a statement. Here are three statements:

```
x = 7;
i++;
printf("Hello\n");
```

Expressions always have a type, but statements never do. (And like in Java, the type of some expressions is **void**.)

The **if-else** statement

The general form of a C **if-else** statement is the same as Java:

```
if (expression)
    statement1
else
    statement2
```

expression is evaluated and if the value produced is non-zero then *statement1* is executed. Otherwise, *statement2* is executed.

There's an **else**-less **if**, too:

```
if (expression)
    statement
```

Example:

```
if ('x' + 3.75e50)
    printf("works!\n");
```

Compound statements

A *compound statement* groups several statements into a single statement.

The general form of a compound statement is:

```
{
    statement1
    statement2
    ...
    statementN
}
```

Example:

```
if (count != 0) {
    printf("Average item weight is %g pounds\n", total_weight / count);
    printf("Average length is %g feet\n", total_length / count);
    }
else
    printf("No items processed!\n");
```

Some companies have *coding standards* that require all control structures to use compound statements. A "commit" of the above might be refused automatically!

The **while** statement

General form:

```
while (expression)
    statement
```

C's **while** is just like Java's, with the variation you'd expect: ***statement*** is executed while ***expression*** yields a non-zero value.

Problem: Using **while**, write an infinite loop in C.

```
while (1) {
    ...
    }
```

Problem: Given an **int n**, print "**Hello!**" **n** times. Restriction: No assignments or relational operators!

```
while (n--)
    printf("Hello!\n");
```

Quick: Is it correct or OBO?

The **do-while** statement

C has a **do-while**, also conditionalized on a non-zero value:

```
do
    statement
while (expression) ;
```

Example:

```
int n = 3;
do {
   printf("%d\n", n);
} while (--n);
```

What's a hazard of the above **do-while**?

What if **n** *is zero?*

Are the braces needed above?

Byte-by-byte I/O

UNIX files are just a series of bytes

All UNIX files can be considered to be a simple, continuous series of bytes. A byte is eight bits on most modern systems, providing 256 distinct values.

As we've seen, **od** can be used to display a file's bytes as a sequence of integers:

```
% cat text2
abc
0

% od -td1 -Ad text2
0000000   97   98   99   10   48   10
```

od can show bytes as signed (-128 to 127) or unsigned (0 to 255) values:

```
% head -c12 /dev/urandom > bytes

% od -td1 bytes
0000000 -118  79  -10  21 -107 102  86  24 10  41  52 102

% od -tu1 bytes
0000000  138 79   246 21  149  102  86  24 10  41  52 102
```

The **getchar()** function

One way to read a file one byte at a time is with the function **getchar()**, defined by the C11 standard:

7.21.7.6 The `getchar` function

Synopsis

```
#include <stdio.h>
int getchar(void);
```

Description

The **`getchar`** function is equivalent to **`getc`** with the argument **`stdin`**.

Returns

The **`getchar`** function returns the next character from the input stream pointed to by **`stdin`**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **`getchar`** returns **`EOF`**. If a read error occurs, the error indicator for the stream is set and **`getchar`** returns **`EOF`**.

EOF is a macro defined by including **stdio.h**. We'll talk about **EOF** soon.

There are man pages for the C library functions. Try **man getchar**.

dumpbytes.c

Here's a program that reads bytes from standard input and prints the value of each byte both as an integer and as a character:

```
% cat dumpbytes.c
#include <stdio.h>
int main(int argc, char *args[])
{
   int byte_num = 0, c;

   while ((c = getchar()) != EOF)
      printf("%04d: %3d (%c)\n",
            byte_num++, c, c);
   return 0;
}

% gcc -o dumpbytes dumpbytes.c
```

Execution:

```
% cat text2
abc
0
% dumpbytes < text2
0000:  97 (a)
0001:  98 (b)
0002:  99 (c)
0003:  10 (
)
0004:  48 (0)
0005:  10 (
)
%
```

dumpbytes.c, continued

```
% echo -n x | dumpbytes
0000: 120 (x)
%
% echo -n x | dumpbytes | dumpbytes
0000:  48 (0)
0001:  48 (0)
0002:  48 (0)
0003:  48 (0)
0004:  58 (:)
0005:  32 ( )
0006:  49 (1)
0007:  50 (2)
0008:  48 (0)
0009:  32 ( )
0010:  40 (()
0011: 120 (x)
0012:  41 ())
0013:  10 (
)
%
```

For reference, **dumpbytes.c**:

```
#include <stdio.h>
int main(int argc, char *args[])
{
    int byte_num = 0, c;

    while ((c = getchar()) != EOF)
        printf("%04d: %3d (%c)\n",
        byte_num++, c, c);
    return 0;
}
```

A line and character counter

Here's a program that counts lines and characters on standard input. Note that it counts lines by simply counting newlines.

```
#include <stdio.h>
int main(int argc, char *args[]) // clc.c
{
    int c, num_lines = 0, num_chars = 0;

    while ((c = getchar()) != EOF) {
        num_chars++;
        if (c == '\n')
            num_lines++;
        }

    printf("%d lines, %d characters\n",
            num_lines, num_chars);
}
```

Usage:

```
% cat text
A
test
right here
% clc < text
3 lines, 18 characters
```

Is **clc** too chatty? Should it output just "**3 18**"?

End-of-file

Imagine that Bob is reading a list of words to Carole over the phone. Carole is repeating the words to Alice, who's writing them down. Here's what Carole says:

"gondola
version
impious
data
viking
oops
lost
him"

What was the last word that Bob read to Carole?

Carole and Alice can talk further to settle the question but programs need a simple way to distinguish data from the end of the data.

End-of-file, continued

The problem:

How can we distinguish data from the end of the data?

One approach is to use a *sentinel value*, a value that won't appear in the data.

What's a problem with using a value like **-999999999** as a sentinel value when summing a series of integer values?

*It precludes **-999999999** from being in the input.*

If we're reading **int**s with **Scanner**, what are our options for detecting end of file?

*We can use **hasNextInt()***
*We can catch **NoSuchElementException***
Anything else?

If we're reading integers one per line from standard input using **BufferedReader.readLine()** what would be a good sentinel value other than relying on **readLine()** returning null at end of file?

*Perhaps a line like "**end**".*

End-of-file, continued

Here's interaction with the **ed** editor:

```
% ed
i
one
two
.
1,$p
one
two
w x
8
q
% cat x
one
two
%
```

Does **ed** use a sentinel value?

Yes, a period by itself on a line terminates input mode.

How could we create file like **dots**?

```
% cat dots
.
.
.
%
```

We might enter three **x***'s and then change them to periods with* **1,$s/x/./**

The end-of-file problem for bytes

The problem:

An 8-bit byte read from a file might have any of 256 different values. An 8-bit **`char`** can only hold 256 different values. There's no **`char`** value that's left free to be used as a sentinel value.

Should we recognize one of those 256 characters as an end-of-file sentinel character? Is that what control-D is?

A fact in the mix: ASCII code 25 is called "**`EM`**", for "End of medium".

getchar() and end-of-file

Let's look at the C11 standard's synopsis for **getchar()** again:

Synopsis

```
#include <stdio.h>
int getchar(void);
```

Note that **getchar()** returns an **int**!

The standard goes on to say,

"If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getchar** returns **EOF**."

Recall this input idiom:

```
while ((c = getchar()) != EOF) ...
```

How can we find out what the **EOF** character is?

```
% gcc -E dumpbytes.c | fgrep "c = getchar"
   while ((c = getchar()) != (-1))
```

What's going on?

getchar() and end-of-file, continued

The facts:

If we want to use all 256 byte values there's no value free to be a sentinel.

Despite its name, **getchar()** returns an **int**, not a **char**.

getchar() returns the **int** -1 at end-of-file.

getchar()'s solution for distinguishing data from the end of the data is to return a larger quantity, an **int**, which can hold any of the 256 possible byte values (0-255) as well as a sentinel value, -1.

Imagine a C implementation of Java's **BufferedReader.readLine()** that uses **getchar()**. When **getchar()** returns -1, that code might then return a data structure that represents **null** in Java. (Note: quite a bit simplified!)

Java's **Reader.read()** method returns "The character read, as an integer in the range 0 to 65,535 (`0x00-0xffff`), or -1 if the end of the stream has been reached." The return type of **Reader.read()** is **int**.

The **putchar()** function

The **putchar(int c)** function causes a byte with the value of **c** to be written to standard output. Example:

```
#include <stdio.h>
int main(int argc, char *args[]) // hellobytes.c
{
   putchar(72);
   putchar(101);
   putchar(108);
   putchar(108);
   putchar(111);
   putchar(33);
   putchar(10);
}
```

Execution:

```
% a.out
Hello!
```

The **putchar()** function, continued

Problem: Write **cat0**, a simple version of **cat**. It need only read bytes from standard input and write them to standard output. Examples:

```
% date > x
% cat0 < x
Fri Sep 25 00:03:00 MST 2015
% cat0 > out
just
testing
^D
% cat0 < out
just
testing
%
```

Solution:

```
#include <stdio.h>
int main(int argc, char *args[])
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

Can we reduce it to the following?

```
int main(int argc, char *args[])
{
    int c;
    while (putchar(c = getchar()) != EOF)
        ;
}
```

The **scanf** function

Here is a program that uses **scanf()**—the input counterpart of **printf()**. The **scanf** call below is somewhat like **Scanner.nextInt()**.

```
% cat writebytes.c
#include <stdio.h>
int main(int argc, char *args[])
{
   int byte_value;

   while (scanf("%d", &byte_value) == 1)
      putchar(byte_value);
}
```

Usage:

```
% writebytes >x
48
65 10
% wc -c x
3 x
% dumpbytes < x
0000:  48 (0)
0001:  65 (A)
0002:  10 (
)
%
```

scanf returns the number of values read or **EOF**.
Do **man scanf** for lots more!

The unary **&** operator will be discussed in detail later. For now, simply note that it produces the address of its operand (a variable). That address is passed to **scanf()**, which stores the integer read into that variable. (Try leaving off the **&**.)

More control structures

The **for** statement

C's **for** statement has the same general form as Java's **for**:

```
for (expr1; expr2; expr3)
    statement
```

Just like **while**, the body of the loop is executed if *expr2* yields a non-zero value.

A loop to print the numbers from 1 through 5:

```
for (int i = 1; i <= 5; i++)
   printf("%d\n", i);
```

A loop to print the alphabet backwards:

```
for (char c = 'z'; c >= 'a'; c--)
   putchar(c);
```

Is the alphabet-printing loop portable? (That is, does the C11 standard guarantee it will run correctly on any machine?)

It is not portable. It assumes a character coding system with a contiguous set of lowercase letters in ascending order.

for, continued

A program to print a *truth table* for the && and || operators:

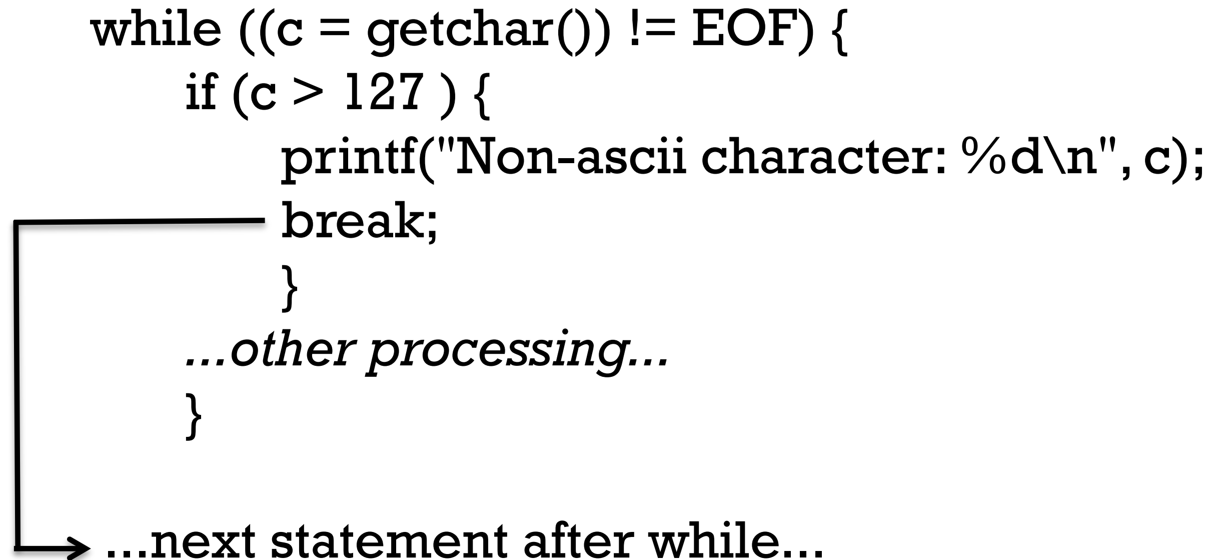
```
int main(int argc, char *args[]) // for2.c
{
  for (int a = 0; a <= 1; a++) {
    for (int b = 0; b <= 1; b++) {
      printf("%d || %d = %d\t", a, b, a||b );
      printf("%d && %d = %d\n\n", a, b, a&&b );
      }
    }
}
```

Output:

```
% gcc for2.c && a
0 || 0 = 0    0 && 0 = 0

0 || 1 = 1    0 && 1 = 0

1 || 0 = 1    1 && 0 = 0

1 || 1 = 1    1 && 1 = 1
```

The **break** statement

In the common case, the **break** statement in C is just like Java's **break**: it causes an immediate exit from the enclosing loop. Example:



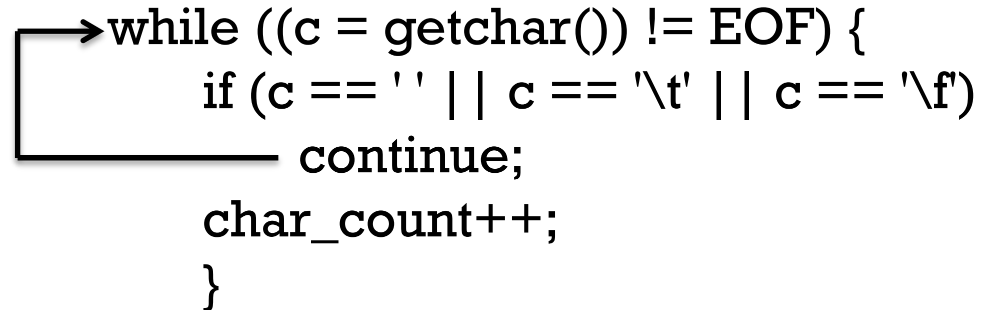
```
while ((c = getchar()) != EOF) {
    if (c > 127 ) {
        printf("Non-ascii character: %d\n", c);
        break;
        }
    ...other processing...
    }

...next statement after while...
```

C <u>does not</u> provide a "labeled **break**" like Java's but the same effect can be achieved with the **goto** statement, which we'll see soon.

The **continue** statement

C's **continue** skips the remainder of a loop iteration and continues with the next iteration of the loop.

In the following example, blanks, tabs, and form-feeds are not included in a character count:



```
while ((c = getchar()) != EOF) {
        if (c == ' ' || c == '\t' || c == '\f')
            continue;
        char_count++;
        }
```

C does not provide a labeled **continue** like Java's but the same result can be achieved with goto.

The goto statement

The goto statement causes control to transfer to the statement with the specified label.

Here's an old-fashioned way to print the numbers from 1 through 10:

```
int main(int argc, char *args[])  // goto1.c
{
            int i = 1;
  top:      printf("%d\n", i++);
            if (i > 10)
                goto done;
            goto top;

  done: printf("All done!\n");
}
```

Before modern control structures evolved it was common to write code such as the above. And, in assembly language, you've only got "branches" and "jumps".

Speculate: Should we teach "goto" first and control structures later?

goto, continued

One modern use of goto is to cleanly exit from deeply nested loops. Java provides a labeled break for that purpose. In C, a goto is reasonable alternative:

```
for (...) {
    while ( ... ) {
        for ( ... )  {
            while (...) {
                ...
                if (...)
                    goto recycle;
                ...
                }
            }
        }
    recycle:
        f();
    }
```

The only other option, introducing a set of flags and tests, often produces code that is hard to reason about.

The goto statement is often handy when a program is generating a C program.

Use of goto in other situations is generally frowned upon.

The **switch** statement

Here's a program that uses a **switch** to count vowels and non-vowels:

```
int main(int argc, char *args[])
{
    int c, num_vowels = 0, num_others = 0;

    while ((c = getchar()) != EOF) {
        switch (c) {
            case 'a': case 'A':
            case 'e': case 'E':
            case 'i': case 'I':
            case 'o': case 'O':
            case 'u': case 'U':
                num_vowels++;
                break;
            default:
                num_others++;
                break;
            }
        }
    printf("%d vowels, %d others\n", num_vowels, num_others);
}
```

An interesting **switch**-based construction that's permitted in C but not in Java is known as "Duff's Device", invented by Tom Duff while at Lucasfilm.

switch, continued

C's **switch** is very similar to Java's **switch**—it selects a block of code to execute based on a controlling expression. Here is a simplified general form:

```
switch (controlling_expr) {
    case const_expr1:
        stmt_block1
        break;
    case const_expr2:
        stmt_block2
        break;
        ...
    default:
        dflt_block
        break;
        }
```

Simplified rules: (6.8.4.2 in C11 has the details)

- *controlling_expr* must produce an integer
- Each *const_expr* must be an constant of integer type and not be duplicated in another case specification.
- A **stmt_block** can have any number of **case**s.
- **default:** can appear only once.
- Execution "falls through" to the next case if there's no **break**.
- If the control expression matches no case and there's no default specified, nothing is executed.

Functions

Function basics

All executable code in a C program is contained in *functions*.

Here's the general form of a function definition in C:

```
return_type function_name(parameters)
{
    ...any mix of declarations and statements...
}
```

Example:

```
int add(int a, int b)
{
   int sum;

   sum = a + b;
   return sum;
}
```

As in Java, the **return** statement terminates execution of a function and produces the specified value as the result of the function invocation.

Function basics, continued

At hand:

```
int add(int a, int b)
{
   int sum;

   sum = a + b;
   return sum;
}
```

A shorter equivalent:

```
int add(int a, int b)
{
   int sum = a + b;

   return sum;
}
```

Even shorter:

```
int add(int a, int b)
{
   return a + b;
}
```

Function basics, continued

Let's add in a **main** with some calls to **add**:

```
#include <stdio.h>
#include <limits.h>
#include "352.h"

int add(int a, int b) // fcn1.c
{
   int sum;
   sum = a + b;
   return sum;
}

int main(int argc, char *args[])
{
   int r1 = add(3,4);
   printf("r1 is %d\n", r1);
   int r2 = add(add(r1+5,20),add(r1-10,add(r1*r1,50)));
   printf("r2 is %d\n", r2);
   iprint(add(INT_MIN,INT_MAX));
}
```

Execution:

```
% gcc fcn1.c && a.out
r1 is 7
r2 is 128
add(INT_MIN,INT_MAX) is -1
```

Sidebar: *translation units*

The C11 standard says that the body of code that is output by the preprocessor is known as a *translation unit*.

Let's put the translation unit for **fcn1.c** into **x**:

```
% gcc -E fcn1.c > x
```

Let's see how many lines, words and characters are in the translation unit for **fcn1.c**:

```
% wc x
 723  1789 14870 x
```

Let's see on which lines the word "**add**" appears in the translation unit for **fcn1.c**.

```
% fgrep -n -w add x
707:int add(int a, int b)
718:    int r1 = add(3,4);
720:    int r2 = add(add(r1+5,20),add(r1-10,add(r1*r1,50)));
%d\n", add((-2147483647 - 1),2147483647));
```

Important: The translation unit is <u>not</u> the file **x**; <u>it's the lines in the file **x**</u>.

Type checking for function calls

Here's **add** again:

```
int add(int a, int b)
{
   return a + b;
}
```

Here are some invalid uses of **add**:

```
add(3)       // too few arguments
add(3,4,5)   // too many arguments
add(3,"4")   // second argument has wrong type
```

The C compiler is required to detect errors such as the above when the code is being compiled.

Java also turns up such errors at compile time but in dynamically typed languages like Ruby and Python, the above misuses of **add** wouldn't turn up until the program was run.

Type-checking for function calls, continued

A simplified rule: For the compiler to confirm that a function **f** is being properly used, either (1) the definition of **f** or (2) a *prototype* for **f** must appear in a translation unit before the first call of **f** in that unit.

Here's a skeleton of **fcn1.c**:

```
int add(int a, int b) // this is a definition of add
{
   return a + b;
}

int main(int argc, char *args[])
{
   int r1 = add(3,4);
   ...
}
```

The code above satisfies (1): **add** is defined before its first call, in **main**.

A call such as **add(3)** in **main** would noted as an error because the definition of **add** has two arguments.

Type-checking for function calls, continued

At hand:

A simplified rule: For the compiler to confirm that a function **f** is being used as expected, either (1) the definition of **f** or (2) a *prototype* for **f** must appear in a translation unit before the first call of **f** in that unit.

In the following code the definition for **add** appears after main, but a prototype for **add** appears before main. That satisfies (2) above.

```
int add(int a, int b);                   // prototype for add

int main(int argc, char *args[])         // fcn1a.c
{ int r1 = add(3,4); ... }

int add(int a, int b) { return a + b; }  // definition for add
```

A prototype specifies a function's return type and argument types but has no body.

Type-checking for function calls, continued

At hand:

A simplified rule: For the compiler to confirm that a function **f** is being used as expected, either (1) the definition of **f** or (2) a *prototype* for **f** must appear in a translation unit before the first call of **f** in that unit.

Lots of old C code is still in service. To accommodate that old code (and old coders) there is a far more complicated set of type-checking rules that is actually used for function calls.

However, we'll write code that heeds our simplified rule, and fix any warnings that arise when we don't get it right.

Note that changing **-Wall** to **-Werror** in your **gcc** alias causes all warnings to be treated as errors. I hesitate to recommend **-Werror** in your **gcc** alias since that breaks an occasional interesting experiment, but the Tester will use **-Werror** for C programs.

A prototype is a promise

In addition to describing the interface of a function, a prototype is a promise that somewhere there exists a matching <u>definition</u> of the function. If no definition is found, an error is produced:

```
% cat fcn1b.c
#include <stdio.h>
...
int add(int a, int b); // "prototype" for add

int main(int argc, char *args[])
{
   iprint(add(3,4));
}

% gcc fcn1b.c
/tmp/cckFCOPS.o: In function `main':
/home/whm/cw/c/fcn1b.c:8: undefined reference to `add'
collect2: ld returned 1 exit status
```

A prototype is a promise, continued

Let's see where **printf** appears in the translation unit:

```
% gcc -E fcn1.c | fgrep -nw printf
445:extern int printf (__const char *__restrict __format, ...);
719:    printf("r1 is %d\n", r1);
721:    printf("r2 is %d\n", r2);
722:    printf("add(INT_MIN,INT_MAX)" " is %d\n",
add((-2147483647 - 1),2147483647));
```

The prototype for **printf** at line 445 of the translation unit allows the compiler to confirm that the calls to **printf** at lines 719, 721, and 722 are correct.

Note that there's no definition for **printf** in the translation unit. **printf** comes from the C library. **gcc**'s last step is "linking", done by **ld(1)**, and **ld** produces an executable that either contains the code for **printf** or references the code in a "shared library".

A curiosity

Consider this:

```
% cat fcn2.c
int f();

int main(int argc, char *args[])
{
   f(3);
}

int f() { ... }

% gcc fcn2.c
%
```

Why doesn't **gcc** indicate that there's a mismatch between **int f()** and the call **f(3)**?

A curiosity, continued

At hand—the following produces no errors.

```
int f();

int main(...) { f(3); }

int f() { ... }
```

Why doesn't **gcc** indicate that there's a mismatch in the number of arguments?

From C11:

> A *function prototype* is a declaration of a function that declares the types of its parameters.

The line **int add();** is a function declaration but it is not a function prototype!

This distinction rises from the standard's support for legacy code.

Note that violations of our simplified "rule" on slide 134 are not seen as errors by **gcc**! That "rule" is really just a recommended practice for us.

A curiosity, continued

A prototype for a function that has no parameters is specified like this:

```
int f(void);
```

Here's a fully correct version:

```
int f(void);

int main(...) { f(); }  // f(3) will produce an error

int f(void) { ... }
```

I was surprised to see that following compiles without an error or warning:

```
int f() { ... }
int main(...) { f(3); }
```

gcc does generate an error for this:

```
int f(void) { ... }
int main(...) { f(3); }
```

Source file layout

It is common to put **#include** directives first in a source file, followed by prototypes for functions that appear later in the file, followed by function definitions. Example:

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

int next_word_len(...);
void add_words(...);

int main(int argc, char *args[]) { ...code... }

void add_words(...) { ...code... }

int next_word_len(...) { ...code... }
```

Note that the prototypes are "at file scope", i.e., they are not contained in a function.

Not all "functions" are functions

Despite the name, some "functions" don't return a value. A return type of **void** indicates that the "function" yields no value. Example:

```
void printN(int n, char c)
{
    while (n--)
        putchar(c);
}
```

Not all "functions" are functions, continued

If a function's return type is **void**, no value is specified in the **return** statement:

```
void f(int n)
{
    if (g(n))
        return;
    ...
}
```

There is an implicit return at the end of every routine.

The terms "routine" and "subroutine" are synonyms for "function" and are often used to communicate the possibility that a "function" doesn't necessarily return a value.

<u>It is incorrect to refer to a C function as a "method".</u>

Recursive functions

A function that calls itself, either directly or indirectly, is said to be *recursive*. Like most modern languages, C supports recursive functions.

Here's a recursive function that prints the digits of a whole number:

```
void pdigits(int n)
{
   if (n == 0)
      return;
   else {
      pdigits(n / 10);
      printf("%d\n", n%10);
      }
}

int main(int argc, char *args[])
{
   pdigits(47819);
}
```

Output:

```
4
7
8
1
9
```

About **pdgits** I would say,
The recursion first "bottoms out" and prints the digits on the way up.

(No!) Overloading

In Java, methods can be *overloaded*: any number of methods can have the same name as long as their *signatures* differ by the number of arguments and/or argument type.

Example: (in Java)

```
public class X {
   int cube(int n) ...
   float cube(float n) ...
   double cube(double n) ...

   void f(int a) ...
   void f(int a, char b) ...
   }
```

Overloading is common in Java's library classes. Two examples:

```
% javap java.lang.String | grep valueOf | wc -l
9

% javap java.io.PrintStream | grep println | wc -l
10
```

(No!) Overloading, continued

Unfortunately, C does not support overloading of functions. There can only be one globally-accessible function with a given name in an entire program.

One workaround is to use more descriptive names:

```
int cube(int n);
float fcube(float n);
double dcube(double n);

void f(int a);
void f2(int a, char b);
```

Try "**man abs**" and "**man fabs**".

We'll later see some alternatives provided by the preprocessor and "static" functions.

Contrast: C++ does allow overloaded functions

Local variables

A variable declared inside a function definition is called a local variable.

It is said that a variable "lives" when storage is guaranteed to be reserved for it.

The lifetime of a local variable is from the time of entry into the "block" containing the variable to the time that execution of code in the block ends.

One type of block is the body of a function. Consider this function:

```
void f()
{
   int i;

   ...executable code...

   int j;

   ...more code...
}
```

Two local variables, **i** and **j**, come into existence when **f()** is called. **i** and **j** "live" until the function returns.

If **f** is called many times, **i** and **j** come to life and then die many times.

If **f** is recursive, there may be many instances of **i** and **j** alive at the same time—each active call will have its own copies of them.

Local variables, continued

Local variables are said to have ***automatic storage duration***, one of four types of "storage durations" by the C11 standard. (The other three are *static*, *allocated*, and *thread*.)

In a typical C implementation, variables with automatic storage duration reside on the execution stack but the standard does not require that a stack be present. The standard simply requires that space be reserved somewhere when the local variables are live.

A note on nomenclature:

The standard does not contain the term "local variable". Such variables are referred to as "objects with automatic storage duration".

An in-between term is *automatic variable*.

Local variables, continued

Automatic variables can also be declared in a compound statement. (A compound statement is another type of block.)

In C, inner declarations hide outer declarations.

```
int main(int argc, char *args[])
{
    int a = 1, b = 2;
    if (...) {
        int c = 3, d = 4;
        // Can use a, b, c, d here

        if (...) {
            int a = 5; // hides outer a
            int c = 6; // hides outer c
           // a is 5, b is 2, c is 6, d is 4
           }

        // What are the values of a, b, c, and d here?
        }

    // What are the values of a, b, c, and d here?
}
```

Note that C allows hiding that Java does not: The inner declaration of **a** and **c** would be considered an error in Java.

Inadvertently hiding a local variable or parameter can produce a hard-to-find bug.

An "(un)lucky" program

Fact 1:
The value of an automatic variable is not guaranteed to survive between lifetimes.

Fact 2:
Because storage for automatic variables is reused and variables are not initialized by default, <u>an automatic variable might happen to have a "leftover" value.</u>

Example:

```
int f(int i);
int main(int argc, char *args[]) // local1.c
{
    int a = f(1);
    int b = f(0);

    printf("a = %d, b = %d\n", a, b);
}

int f(int i)
{
    int v;

    if (i == 1) v = 10;

    return v;
}
```

Output:

```
a = 10, b = 10
```

Even though <u>**v** is never assigned a value in the second call of **f**</u>, a reasonable value, 10, is returned!

gcc does not produce a warning!

An "(un)lucky" program, continued

```
int f(int i);
int main(int argc, char *args[]) // local1.c
{
    int a = f(1);
    int b = f(0);

    printf("a = %d, b = %d\n", a, b);
}

int f(int i)
{
    int v;

    if (i == 1) v = 10;   // Imagine a block of complex code here, not
                          // just "v =10". It might not be obvious that v is
                          // never given a value.
    return v;
}
```

I call this a "lucky program"—its behavior is undefined by the standard but it produces a reasonable-looking result.

In fact, this behavior is most unlucky because a change can produce a malfunction in seemingly unrelated code.

An "(un)lucky" program, continued

Let's add a trivial function **g** and call it between the two calls to **f**:

```
int f(int i);
void g() { int a = 99; }
int main(int argc, char *args[])
{
    int a = f(1);
    g();
    int b = f(0);

    printf("a = %d, b = %d\n",
        a, b);
}

int f(int i)
{
    int v;
    if (i == 1)
        v = 10;
    return v;
}
```

Output:

```
a = 10, b = 99
```

In real-world terms, a C program that has worked "perfectly" for years might break due to insertion of a seemingly innocuous function call.

Similarly, simply removing an unused variable might cause a "lucky" alignment to disappear.

Is there a possibility of a bug like this in Java? How can bugs like this be prevented in C?

Global variables

A variable declaration can appear outside of a function. Such a variable is called a *global variable* or an *external variable*.

A global variable can be accessed or modified by any function.

Here is a **cat**-like program that uses a global variable to hold the character currently being processed: (**glob1.c**)

```
int c;  // c is a global variable

void gchar() { c = getchar(); }
void pchar() { putchar(c); }
int main(int argc, char *args[]) {
   while (1) {
      gchar();
      if (c == -1) break;
      pchar();
      }
}
```

The declaration of a global variable must precede any use of the variable.

Note: This is not a good use of a global variable; it simply shows the mechanics.

What happens if a we intend a function to a have a local **int c;** but forget the declaration?

It will use the global variable!

Global variables, continued

Using the "One Big Class" analogy, a global variable in C is analogous to a class variable in Java.

Here's a Java approximation of the preceding C program.

```
public class cat0 {
    static int c;
    public static void main(String args[ ]) throws Exception {
        while (true) {
            gchar();
            if (c == -1)
                break;
            pchar();
            }
        }
    static void gchar() throws Exception { c = System.in.read(); }
    static void pchar() {  System.out.write(c); }
}
```

Global variables, continued

One situation in which a global variable is appropriate is when a variable is used by so many routines that it would be cumbersome to propagate it as a parameter.

Here is a skeletal example that uses a global variable **debug** to control the generation of debugging output.

What's an alternative to having global **debug** variable?

Add a **debug** parameter to all routines that need to see if debugging is on.

If **f()** calls **g()**, **g()** calls **h()** and **h()** needs **debug**, then **debug** needs to be passed to/through all those routines.

```
int debug = 0; // global variable
int main(int argc, char *argv[ ]) {
   if (/* -d option specified */)
      debug = 1;
   ...
}

void f(...)
{
   if (debug)
      printf(...);
   ...
}

void g(...)
{
   if (debug)
      show_mapping();
   ...
}
```

Global variables, continued

Global variables have *static storage duration*. Variables with static storage duration are initialized before execution of the program begins.

An implication of this rule is that an initializer for a global must be an expression that can be evaluated at compile time. Here are some suitable initializations:

```
int debug = 0;                   // OK: Just a constant.

int secs_in_day = 24*60*60;   // OK: Expression uses only constants.

int s2 = secs_in_day * 2;       // Error: Uses a variable.

int x = h();                     // Error: Calls a function. (Assuming h()
                                 //  is not a macro.)
```

What's the benefit of guaranteed initialization before execution begins?

Global variables, continued

Imagine a collection of library functions that require initialization of a data structure before any of them can be used, but there's no guarantee which might be called first. Here's how the library might solve the problem:

```
int library_intialized = 0;
void init_library()
{
    if (!library_intialized) {
        ...library initialization code...
        library_intialized = 1;
        }
}

double f(...)
{
    init_library();
    ...f's computations...
}

double g(...)
{
    init_library();
    ...g's computations...
}
```

If we couldn't count on **library_initialized** being zero we'd need to require the user to call **init_library** before using any of the functions. (But would that really be a burden?)

Recall that in our I/O examples we didn't need to do anything special before calling **getchar()**. Convenient, huh?

Global variables, continued

Unlike local variables, the standard specifies that variables with static storage duration (such as globals) are initialized to zero (in lieu of explicit initialization).

The standard guarantees that the program below will output "**x = 0**" even though there's no initializing value specified for **x**.

```
int x;      // global
int main(int argc, char *args[])
{
   printf("x = %d\n", x);
}
```

Globals that are **float** or **double** are similarly set to an arithmetic zero. Various types we haven't seen yet are similarly set to a type-specific "zero".

We'll see later see some cases where this zero-initialization guarantee provides significant performance improvements with arrays.

Static variables

If a variable inside a function is declared as **static**, its value survives across function invocations. Example:

```
int gen_id() // static3.c
{
    static int next_id = 1000;
    return next_id++;
}

int main(int argc, char *args[])
{
    iprint(gen_id());
    iprint(gen_id());
    iprint(gen_id());
}
```

Output:

```
gen_id() is 1000
gen_id() is 1001
gen_id() is 1002
```

Would a global variable **next_id** work just as well?

With a static local variable we be sure that only the code in the function has access to the variable. (Not counting a wild memory reference elsewhere!)

Static variables, continued

In essence, a static local is a global variable that can be referenced by only one function. Along with limiting the variables scope to the function it is self-documenting code that clearly indicates that the variable is associated with the function.

Conceptually, a static local is initialized on the first call of the function that contains it.

Static locals have static storage duration, just like globals, and follow the same initialization rules:

- An initializer for a static variable must be an expression that can be evaluated at compile-time.
- If no initializer is specified, the initial value is a type-appropriate zero.

There is no counterpart in Java for static local variables.

Java methods vs. C functions

A little Java review:

A Java method is an encapsulation of code that has access to some number of *fields* that are implicitly associated with it by virtue of being in the same class.

- **static** methods have access to static fields, which are often called *class variables*.

- non-**static** methods have access to both static and non-static fields. Non-static fields are often called *instance variables*.

Like a Java method, a C function is an encapsulation of code.

Unlike a Java method, there is no implicit association of data with a C function, other than the function's local variables.

Java methods vs. C functions, continued

Java:

Programs are composed of classes, which associate code (methods) and data (fields).

Access to the fields of a class can be controlled with access modifiers like `public` and `private`.

C:

Programs are composed of functions that have access to all "global" variables and file-scope variables in the same file.

A little `gdb`

gdb basics

gdb is the GNU Debugger.

There are two common uses of **gdb**:

1) **gdb** can be used to control the execution of a program, setting breakpoints, executing line by line, and examining variables as desired.

2) **gdb** can be used to examine a "core dump" (a **core** file) and determine the state of the program when the operating system terminated its execution.

gdb relies heavily on "debugging information" that by default is <u>not</u> included in an executable.

gcc's -g option directs **gcc** to include debugging information in an executable.

Our **gcc** alias includes **-g**.

gdb basics, continued

gdb has very complete built-in documentation. Just typing "help" shows the various categories of gdb commands:

```
% gdb -q    # -q for "quiet" startup
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the
program
user-defined -- User-defined commands
...more...
(gdb)
```

A gdb session

We'll be working with examples from **fall15/c/gdb**.

loop1 is an executable for **loop1.c**, created with **gcc -o loop1 loop1.c**.

To run a program under the control of **gdb**, simply specify the executable on the command line:

```
% gdb -q loop1
Reading symbols from /p1/hw/whm/352/loop1...done.
(gdb)
```

When **gdb** starts up, it readies the target executable for execution but **gdb** does not start it running.

The **(gdb)** prompt indicates that **gdb** is ready to proceed.

A gdb session

gdb's **list** command is used to display the source code associated with the executable. If `loop1.c` were not in the same directory as `loop1`, we'd need to take an additional step to tell **gdb** the location of `loop1.c`.

```
 % gdb -q loop1
Reading symbols from /p1/hw/whm/352/loop1...done.
(gdb) list
2
3       void print(int n)
4       {
5           if (n % 2 == 1)
6               printf("The value %d is odd\n", n);
7       }
8
9       int main(int argc, char *args[])
10      {
11          int i = 1, N = 10;
```

A gdb session, continued

To see more of the program we might press **RET** (ENTER), which causes the last command (**list**, in this case) to be repeated:

```
(gdb) RET
12
13          while (i <= N) {
14              print(i);
15              i++;
16              }
17
18          printf("All done!\n");
19      }
(gdb) RET
Line number 20 out of range; loop1.c has 19 lines.
(gdb)
```

The command **list 14** shows a "window" of lines around line 14. Try it!

A gdb session, continued

Like most **gdb** commands, **list** has several forms:

```
(gdb) help list
List specified function or line.
With no argument, lists ten more lines after or around previous listing.
"list -" lists the ten lines before a previous ten-line listing.
One argument specifies a line, and ten lines are listed around that line.
Two arguments with comma between specify starting and ending lines to list.
Lines can be specified in these ways:
  LINENUM, to list around that line in current file,
  FILE:LINENUM, to list around that line in that file,
  FUNCTION, to list around beginning of that function,
  FILE:FUNCTION, to distinguish among like-named static functions.
  *ADDRESS, to list around the line containing that address.
With two args if one is empty it stands for ten lines away from the other arg.
```

Examples:

```
list 3,30          // list lines 3 through 30
list print         // list code around the beginning of the 'print' function
list scanner.c:30,90   // list lines 30 through 90 in scanner.c
```

The **run** command causes the target program to be run:

```
(gdb) run
Starting program: /p1/hw/whm/352/loop1
The value 1 is odd
The value 3 is odd
The value 5 is odd
The value 7 is odd
The value 9 is odd
All done!
[Inferior 1 (process 21093) exited normally]
(gdb)
```

The program ran to completion without encountering any errors.

A **gdb** session, continued

A common reason to run a program with **gdb** is to pause execution at a designated point (or points). The **break** command sets a "breakpoint".

If a function name is specified as the argument of **break**, a breakpoint is established at the first line of the function that has executable code.

Setting a breakpoint in **main** gives us a chance to get control as soon as the program starts running:

```
(gdb) break main
Breakpoint 1 at 0x40058e: file loop1.c, line 11.
```

A subsequent **run** command starts the program running. The breakpoint in **main** is hit immediately, and the first line with executable code is displayed:

```
(gdb) run
Starting program: /p1/hw/whm/352/loop1

Breakpoint 1, main (argc=1, args=0x7fffffffd6c8) at loop1.c:11
11          int i = 1, N = 10;
```

A **gdb** session, continued

For reference:

```
(gdb) run
Starting program: /p1/hw/whm/352/loop1

Breakpoint 1, main (argc=1, args=0x7fffffffd6c8) at loop1.c:11
11          int i = 1, N = 10;
```

Variables can be examined using the **print** command.

```
(gdb) print i
$1 = 0
(gdb) print N
$2 = 0
```

Why are the values of **i** and **N** wrong?

The initializations haven't been done yet.

A gdb session, continued

For reference:

```
Breakpoint 1, main (argc=1, args=0x7fffffffd6c8) at loop1.c:11
11        int i = 1, N = 10;
(gdb) print i
$1 = 0
(gdb) print N
$2 = 0
```

The **next** command causes execution of one statement:

```
(gdb) next
13        while (i <= N) {
(gdb) print i
$3 = 1
(gdb) print N
$4 = 10
```

A gdb session, continued

For reference:

```
(gdb) next
13        while (i <= N) {
```

The **next** command "steps over" function calls.

```
(gdb) n
14            print(i);
(gdb) RET (ENTER, repeats the last command, n(ext))
The value 1 is odd
15            i++;
(gdb) RET
13        while (i <= N) {
(gdb) RET
14            print(i);
(gdb) RET
15            i++;
(gdb) p i
$5 = 2
(gdb) n
13        while (i <= N) {
(gdb) RET
14            print(i);
```

A **gdb** session, continued

The **step** command "steps into" function calls, rather than over them.

```
(gdb) n
13          while (i <= N) {
(gdb) RET
14              print(i);
(gdb) step
print (n=4) at loop1.c:5
5           if (n % 2 == 1)
```

The above shows that we've stepped into **print**, the parameter **n** is **4**, and we're ready to execute line 5. Let's look around with **list**:

```
(gdb) list 2,7
2
3       void print(int n)
4       {
5           if (n % 2 == 1)
6               printf("The value %d is odd\n", n);
7       }
```

A gdb session, continued

The **bt** command (also known as **where**) shows a "back trace" of the stack:

```
(gdb) bt
#0  print (n=4) at loop1.c:5
#1  0x00000000004005a8 in main (argc=1, args=0x7fffffffd6c8) at
loop1.c:14
```

bt shows that we're in print, at line 5, and that print was called from line 14.

```
(gdb) list 13,15
13        while (i <= N) {
14            print(i);
15            i++;
```

A gdb session, continued

The print command is not limited to variables; its argument is an arbitrary expression:

```
(gdb) p n
$6 = 4
(gdb) p n % 3
$7 = 1
(gdb) p n % 2 == 1
$8 = 0
(gdb) p print(n*2+1)     // Note: this calls print
The value 9 is odd
$9 = void
(gdb) p i
No symbol "i" in current context.
```

i was not found because **i** is a local variable in main and our current context is print.

A gdb session, continued

At hand: **i** is not found because **i** is a local variable in **main** and our current context is **print**.

```
(gdb) p i
No symbol "i" in current context.
```

The **up** and **down** commands can be used to shift the focus up or down the call stack:

```
(gdb) bt
#0  print (n=4) at loop1.c:5
#1  0x00000000004005a8 in main (argc=1, args=0x7fffffffd6c8) at
loop1.c:14
(gdb) up
#1  0x00000000004005a8 in main (argc=1, args=0x7fffffffd6c8) at
loop1.c:14
14          print(i);
(gdb) p i
$10 = 4
(gdb) down
#0  print (n=4) at loop1.c:5
5       if (n % 2 == 1)
(gdb) p n
$11 = 4
```

A gdb session, continued

A break point can be set by line number:

```
(gdb) list 3,7
3       void print(int n)
4       {
5           if (n % 2 == 1)
6               printf("The value %d is odd\n", n);
7       }
(gdb) b 5
Breakpoint 3 at 0x40054f: file loop1.c, line 5.
```

The continue command causes execution to resume and continue until a breakpoint is hit or the program terminates.

```
(gdb) continue
Continuing.

Breakpoint 3, print (n=5) at loop1.c:5
5           if (n % 2 == 1)
(gdb)
Continuing.
The value 5 is odd

Breakpoint 3, print (n=6) at loop1.c:5
5           if (n % 2 == 1)
```

A **gdb** session, continued

A breakpoint can be deleted. "**info break**" (**i b**) lists the current set of breakpoints. **d(elete)** is used to delete breakpoints.

```
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040058e in main at loop1.c:11
        breakpoint already hit 1 time
3       breakpoint     keep y   0x000000000040054f in print at loop1.c:5
        breakpoint already hit 2 times
(gdb) d 3
```

If execution is continued, and no breakpoints are encountered, the program runs to completion.

```
(gdb) c
Continuing.
The value 7 is odd
The value 9 is odd
All done!
[Inferior 1 (process 30292) exited normally]
```

Conditional breakpoints

A breakpoint can be conditionalized:

```
(gdb) b 8 if c == '\n'
Breakpoint 2 at 0x40055c: file dumpbytes.c, line 8.
(gdb) run < 3bytes
0000:  57 (9)

Breakpoint 2, main (argc=1, args=0x7fffffffd698) at dumpbytes.c:8
8           printf("%04d: %3d (%c)\n", byte_num++, c, c);
(gdb) p c
$7 = 10
(gdb) c
Continuing.
0001:  10 (
)
0002:  49 (1)
0003:  48 (0)

Breakpoint 2, main (argc=1, args=0x7fffffffd698) at dumpbytes.c:8
8           printf("%04d: %3d (%c)\n", byte_num++, c, c);
```

I/O issues

If a gdb-controlled program reads from standard input, it waits for the user to enter a line:

```
% gdb -q dumpbytes
Reading symbols from /p1/hw/whm/352/dumpbytes...done.

(gdb) b 7
Breakpoint 1 at 0x40055a: file dumpbytes.c, line 7.

(gdb) run
Starting program: /p1/hw/whm/352/dumpbytes

Breakpoint 1, main (argc=1, args=0x7fffffffd6c8) at dumpbytes.c:7
7         while ((c = getchar()) != EOF)
(gdb) n
x1        NOTE: The user typed "x1 RET"
8            printf("%04d: %3d (%c)\n", byte_num++, c, c);
(gdb) p/c c        (Print using c(har) format)
$1 = 120 'x'
```

I/O issues, continued

The user entered **x1RET**. We're stopped at line 8:

```
8             printf("%04d: %3d (%c)\n", byte_num++, c, c);
(gdb) p/c c
$1 = 120 'x'
```

Let's execute one line with **next**:

```
(gdb) n
0000: 120 (x)     (Output from printf, at line 8)
7          while ((c = getchar()) != EOF)
(gdb)
```

After output from **printf** (above) we're ready to read the next character.

```
(gdb) n
8             printf("%04d: %3d (%c)\n", byte_num++, c, c);
(gdb) n
0001:  49 (1)
7          while ((c = getchar()) != EOF)
```

Why didn't **gdb** pause for input when executing the **getchar()** on line 7?

Because the user has entered three characters: 'x', '1', and newline.

getchar() *returned the next unread character, the '1'.*

I/O issues, continued

gdb allows shell-like redirection with **run**:

```
% gdb -q dumpbytes
(gdb) b 8
(gdb) run < 3bytes

Breakpoint 1, main (argc=1, args=0x7fffffffd698) at dumpbytes.c:8
8           printf("%04d: %3d (%c)\n", byte_num++, c, c);

(gdb) p/c c
$1 = 57 '9'
(gdb) c
Continuing.
0000:  57 (9)

Breakpoint 1, main (argc=1, args=0x7fffffffd698) at dumpbytes.c:8
8           printf("%04d: %3d (%c)\n", byte_num++, c, c);
(gdb) c
Continuing.
0001:  10 (
)

Breakpoint 1, main (argc=1, args=0x7fffffffd698) at dumpbytes.c:8
8           printf("%04d: %3d (%c)\n", byte_num++, c, c);
```

The **commands** command

The **commands** command specifies one or more commands to be executed when a specific breakpoint is hit.

```
(gdb) b 8
Breakpoint 1 at 0x40055c: file dumpbytes.c, line 8.
(gdb) commands 1
Type commands for breakpoint(s) 1, one per line. End with a line
saying just "end".
>p/c c
>p byte_num
>end
(gdb) run < 3bytes
Breakpoint 1, main (argc=1, args=0x7fffffffd698) at dumpbytes.c:8
8           printf("%04d: %3d (%c)\n", byte_num++, c, c);
$1 = 57 '9'
$2 = 0
(gdb) c
Continuing.
0000:  57 (9)

Breakpoint 1, main (argc=1, args=0x7fffffffd698) at dumpbytes.c:8
8           printf("%04d: %3d (%c)\n", byte_num++, c, c);
$3 = 10 '\n'
$4 = 1
```

If **continue** is the last command, execution will continue after executing the breakpoint's commands.

Lots more with gdb

There's a lot more that can be done with **gdb**. A sampling:

- **finish** Run the current function to completion.
- **search** Search source code for a string.
- **whatis** Show the type of an expression.
- **x** Examine memory.

The info command can show information about dozens of things. Two of them:

- **info locals** Shows the value of all local variables in the current function.
- **info reg** Shows the machine registers.

Remember that the **print** command handles arbitrary expression, including function calls. It can also be used to set variables: **p i = 7**

Keep in mind that **RET** repeats the last command—very handy!

Memory layout

Using & to explore memory layout

C's unary **&** operator is relatively unique among programming languages: it produces the memory address of its operand, which is often a variable.

```
int main(int argc, char **argv) // amp1.c
{
    int i; float f; int j;
    psize(&i);
    printf("&argc = %lu (%p)\n",
      &argc, &argc);
    printf("&i = %lu (%p)\n", &i, &i);
    printf("&f = %lu (%p)\n", &f, &f);
    printf("&j = %lu (%p)\n", &j, &j);
}
```

psize(&i) shows that addresses are eight bytes long—64 bits.

The **%lu** format specifier means **long unsigned**; **%p** means pointer. Both expect a 64-bit value.

%lu generates warnings but we'll use it because it's decimal, not hex.

Output:

```
sizeof(&i) is 8
&argc = 140725604223324 (0x7ffd3ba6895c)
&i = 140725604223332 (0x7ffd3ba68964)
&f = 140725604223336 (0x7ffd3ba68968)
&j = 140725604223340 (0x7ffd3ba6896c)
```

Memory layout, continued

```
int main(int argc, char **argv) // amp1a.c
{
    int i; float f; int j;
    printf("&argc = %lu\n", &argc);
    printf("&i = %lu\n", &i);
    printf("&f = %lu\n", &f);
    printf("&j = %lu\n", &j);
}
```

fall15/bin/dstack is an Icon program that looks for strings of the form **&<id> = <number>** and draws an ASCII picture of memory.

```
% amp1a | dstack
```

(**fall15/bin** is in my **PATH**)

```
                   |-------------|
140729361409420    |      j      |
                   |-------------|
140729361409416    |      f      |
                   |-------------|
140729361409412    |      i      |
                   |-------------|
140729361409408    |     ...     |
                   |-------------|
140729361409404    |    argc     |
                   |-------------|
```

Note that **dstack** orders objects so that addresses descend down the page—high memory is on top.

Memory layout, continued

Let's explore a function call.

```
int sum(int a, int b);

int main(int argc, char *args[]) // amp2.c
{
    int i = 10, j = 20;

    printf("&i = %lu, &j = %lu\n",
        &i, &j);
    sum(i, j);
}

int sum(int a, int b)
{
    int result;

    printf("In sum, &a = %lu, &b = %lu\n",
      &a, &b);
    printf("&result = %lu\n", &result);
    result = a + b;
    return result;
}
```

```
% amp2 | dstack -4
        |-------------|
...4860 |      j      |
        |-------------|
...4856 |      i      |
        |-------------|
          <10 words>
        |-------------|
...4812 |   result    |
        |-------------|
          <3 words>
        |-------------|
...4796 |      a      |
        |-------------|
...4792 |      b      |
        |-------------|
```

Because the parameters and local of **sum** are at lower addresses on the stack than **main**, we say that the stack is *down-growing*— it grows towards the low end of memory. This is common.

Memory layout, continued

Another aspect of memory layout is the placement of global variables and static locals.

```
int gi1;      // global variable
float gf1;   //    ditto
int gi2;     //    ditto
int main(...)  // amp3.c
{
    int i;
    static int s;

    paddr(&gi1); // new macro!
    paddr(&gf1);
    paddr(&gi2);
    paddr(&i);
    paddr(&s);
    paddr(&main);
    paddr(&printf);
}
```

```
                 |-------------|
140725062457132  |      i      |
                 |-------------|
                 5181264m words
                 |-------------|
        6295612  |     gi2     |
                 |-------------|
        6295608  |     gi1     |
                 |-------------|
        6295604  |     gf1     |
                 |-------------|
        6295600  |      s      |
                 |-------------|
                 <525002 words>
                 |-------------|
        4195588  |    main     |
                 |-------------|
                   <64 words>
                 |-------------|
        4195328  |   printf    |
                 |-------------|
```

Note that it's possible to get the address of executable code for a function.

Memory layout, continued

Different C compilers may use different stack layouts even though the target architecture is the same. Let's try **clang** on OS X.

```
% clang --version
Apple LLVM version 6.0 (clang-600.0.56) (based on LLVM 3.5svn)
Target: x86_64-apple-darwin14.0.0
Thread model: posix
% clang amp2.c
% a.out | dstack
                  |------------|
140734661251196   |     i      |
                  |------------|
140734661251192   |     j      |
                  |------------|
                    <6 words>
                  |------------|
140734661251164   |     a      |
                  |------------|
140734661251160   |     b      |
                  |------------|
140734661251156   |   result   |
                  |------------|
```

```
% amp2 | dstack  (gcc on lectura)
                  |------------|
140729433561660   |     j      |
                  |------------|
140729433561656   |     i      |
                  |------------|
                    <10 words>
                  |------------|
140729433561612   |   result   |
                  |------------|
                    <3 words>
                  |------------|
140729433561596   |     a      |
                  |------------|
140729433561592   |     b      |
                  |------------|
```

Memory layout—who cares?

Most Java programmers have no knowledge of typical memory layouts in Java virtual machines and are none the worse for it.

The C standard specifies nothing at all about memory layout, and doesn't even require a stack.

Is it important to be cognizant of memory layout in C?

A loose end—many ways to say **main**

We've been writing **main** like this:

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

The C11 standard allows this, too:

```
int main(void)
{
    ...
}
```

The standard also allows this:

```
int main()
{
    ...
}
```

We'll start avoiding the clutter when we're not using **main**'s arguments.

Arrays

Array basics

Like most languages, C supports the concept of an array—a named sequence of values that can be accessed with integer subscripts.

General form of an array declaration:

type name[*number_of_elements*];

Example:

```
int main()
{
   int a[3];
   ...
}
```

Contrast with Java:

```
int a[] = new int[3];
```

`a` is declared to be an array that has space for three `int` values.

Like Java, C uses zero-based array indexing: the first element in `a` is `a[0]`, the second is `a[1]`, and the third is `a[2]`.

An array can be declared as a local, as above, or a global.

Array basics, continued

At hand:

```
int main()
{
   int a[3];
   ...
}
```

<u>In the same function where an array is declared</u> the size of the array <u>in bytes</u> can be queried with **sizeof**:

```
sizeof(a)  // produces 12, assuming sizeof(int) is 4
```

Note that **sizeof** produces the number of bytes. Here's a good way to calculate the number of elements in a local or global array named **a**:

```
sizeof(a) / sizeof(a[0])
```

<u>There is no analog for Java's *array*.length.</u>

As an alternative, how about **sizeof(a) / sizeof(<u>int</u>)** instead?

*If the type of **a** is changed the divisor, **sizeof(a[0])**, is still correct.*

Array basics, continued

C expressions involving array elements are identical to the equivalent expression in Java. A simple example:

```
int main()  // array4.c
{
   int values[5];    // An array of 5 ints

   values[0] = 1;
   values[1] = 2;
   values[2] = values[0] + values[1];
   values[values[2]] = 10;
   values[4] = 20;

for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++)
      printf("values[%d] = %d\n", i, values[i]);
}
```

Output:

```
values[0] = 1
values[1] = 2
values[2] = 3
values[3] = 10
values[4] = 20
```

Exercise: Step through the program with gdb. (The sooner you get comfortable with gdb, the better!)

Array basics, continued

In Java, arrays ALWAYS reside in the heap.

In C, arrays that have automatic storage duration (i.e., are locals) reside on the stack, alongside other local variables. Array elements are always stored in consecutive, ascending memory locations.

```
int main() // array1.c
{
    int i;
    int a[3];
    int j;

    printf("&i = %u\n", &i);
    for (i = 0; i < 3; i++)
        printf("&a[%d] = %u\n",
            i, &a[i]);
    printf("&j = %u\n", &j);
}
```

```
% array1 | dstack -4
           |-----------|
...6796    |     j     |
           |-----------|
...6792    |     i     |
           |-----------|
             <3 words>
           |-----------|
...6776    |   a[2]    |
           |-----------|
...6772    |   a[1]    |
           |-----------|
...6768    |   a[0]    |
           |-----------|
```

Array initialization

Arrays can be initialized by specifying a sequence of values in braces:

```
double vals[5] = {1.23, 1e2, .023, 7.5, 1e-2};

for (i = 0; i < sizeof(vals)/sizeof(vals[0]); i++)
    printf("vals[%d] = %g\n", i, vals[i]);
```

Output:

```
vals[0] = 1.23
vals[1] = 100
vals[2] = 0.023
vals[3] = 7.5
vals[4] = 0.01
```

If initialization is specified, the size can be omitted:

```
short primes[ ] = { 3, 5, 7, 11, 13 };
printf("sizeof(primes) = %d\n", sizeof(primes));
```

Output:

```
sizeof(primes) = 10
```

Array initialization, continued

Previous examples:

```
double vals[5] = {1.23, 1e2, .023, 7.5, 1e-2};

short primes[ ] = { 3, 5, 7, 11, 13 };
```

Given a size specification, if too many initializers are specified it is a compile-time warning or error. If too few initializers are specified, the remaining values are initialized with a type-appropriate zero.

Examples:

```
int a[10] = {10, 2, 4} ;      // initializes a[3] through a[9] with 0

double b[1] = {};             // initializes b[0] with 0.0

int c[2] = {10, 2, 4};        // warning with gcc
```

What's the difference between the following two?

```
int x1[10];
int x2[10] = {};
```

Variable length arrays

The size of a local array can be specified at run-time:

```
void f(int n);  // vla1.c
int main()
{
   f(9);
   f(3);
}
void f(int n)
{
   int a[n];
   printf("sizeof(a) = %zu\n", sizeof(a));
}
```

Output:

```
sizeof(a) = 36
sizeof(a) = 12
```

a is a *variable length array*, or VLA.

Once created, the size of a VLA doesn't vary. The potential variability is between different instantiations (lifetimes) of the array.

VLAs create a special case for **sizeof**—the size of a VLA cannot be computed until the array is created.

Variable length arrays, continued

For reference:

```
void f(int n)
{
   int a[n];
   printf("sizeof(a) = %d\n", sizeof(a));
}
```

Here's what **a** looks like in **gdb**:

```
(gdb) n
11        int a[n];
(gdb) p a
$1 = 0x4005a0
```

We can use an *artificial array* to examine **a**:

```
(gdb) p a[0]@n
$5 = {611092808, 1686719704, -1924603868, 537424685, 630017024,
2099308, 611092812, 1955155176, -1991446492}
```

p a[0]@n directs **gdb** to imagine there's an **n**-element array that starts with **a[0]**, and print it.

Note that the artificial array syntax is **gdb**-specific; it is not a part of C.

Out-of-bounds references

In Java, array accesses are bounds-checked: an out of bounds reference generates an exception.

In C, array accesses are not bounds checked. Example:

```
int main() // arrayoob1.c
{
   int i = 1;
   int a[3];
   int j = 2;

   a[0] = 10;
   a[6] = 50;
   a[7] = 60;

   printf("i = %d, j = %d\n", i, j);
}
```

Output:

```
i = 50, j = 60
```

What happened?

Out-of-bounds references, continued

Let's add some prints for **dstack**:

```
int main() // arrayoob1.c
{
    int i = 1;
    int a[3];
    int j = 2;

    a[0] = 10;
    a[6] = 50;
    a[7] = 60;

    printf("i = %d, j = %d\n", i, j);

    for (i = 0; i <= 7; i++)
        printf("&a[%d] = %lu\n", i, &a[i]);

    paddr(&i);
    paddr(&j);
}
```

Output:

```
i = 50, j = 60
```

```
% arrayoob1|dstack -4
          |------------|
...8476   |   a[7],j   |
          |------------|
...8472   |   a[6],i   |
          |------------|
...8468   |    a[5]    |
          |------------|
...8464   |    a[4]    |
          |------------|
...8460   |    a[3]    |
          |------------|
...8456   |    a[2]    |
          |------------|
...8452   |    a[1]    |
          |------------|
...8448   |    a[0]    |
          |------------|
```

Out-of-bounds references, continued

Let's get rid of **j**:

```
int main() // arrayoob1a.c
{
    int i = 1;
    int a[3];

    a[0] = 10;
    a[6] = 50;
    a[7] = 60;

    printf("i = %d\n", i);

    for (i = 0; i <= 7; i++)
        printf("&a[%d] = %lu\n", i, &a[i]);
    paddr(&i);
}
```

```
% arrayoob1a
i = 1
&a[0] = 140723248558656
&a[1] = 140723248558660
&a[2] = 140723248558664
&a[3] = 140723248558668
&a[4] = 140723248558672
&a[5] = 140723248558676
&a[6] = 140723248558680
&a[7] = 140723248558684
&i = 140723248558668
Segmentation fault (core dumped)
```

With **j** no longer "taking the bullet", more critical data gets hit!

```
% gdb -q arrayoob1a core
...
(gdb) bt
#0  0x0000003c00000032 in ?? ()
#1  0x0000000000000000 in ?? ()
```

Arrays as function parameters

Arrays as parameters

An array can be passed as an argument to a function.

Here's a function that fills an array **a** with **N** copies of **value**:

```
void fill(int a[], int N, int value)
{
   for (int i = 0; i < N; i++)
      a[i] = value;
}
```

Usage:

```
int main()  // arraypar1.c
{
   int vals[5];
   fill(vals, sizeof(vals) / sizeof(vals[0]), 100);
}
```

When done, all five elements of **vals** hold 100.

Why not do **fill(vals, 5, 100)**?

Exercise: Use **gdb** to confirm that **fill()** works.

Arrays as parameters, continued

When an array is an argument, the parameter essentially becomes another name for the caller's array. Example:

```
void fill(int a[], int N, int value)
{
   printf("&a[0] = %lu\n", &a[0]);
   for (int i = 0; i < N; i++)
      a[i] = value;
}
int main() // arraypar1a.c
{
   int vals[5];
   printf("&vals[0] = %lu\n", &vals[0]);
    fill(vals, sizeof(vals) / sizeof(vals[0]), 100);
}
```

Output:

```
&vals[0] = 140736139990368
&a[0] = 140736139990368
```

A simple but accurate view: **a** in **fill** is another name for **vals** in **main**.

Inside **fill**, **a[i]** refers to the same word (an **int**) in memory as **vals[i]**. An assignment to **a[i]** in **fill** changes **vals[i]** in **main**.

Arrays as parameters, continued

Another example:

```
void reverse(int src[], int dest[], int nelems);
void a_print(int a[], int nelems);

int main() // array8.c
{
    int N = 10, vals[N], rvals[N];

    for (int i = 0; i < N; i++)
        vals[i] = i*i;

    a_print(vals, N);
    reverse(vals, rvals, N);
    a_print(rvals, N);
}
```

```
void a_print(int a[], int nelems)
{
    for (int i = 0; i < nelems; i++)
        printf("%2d ", a[i]);
    printf("\n");
}
```

Output:

```
 0  1  4  9 16 25 36 49 64 81
81 64 49 36 25 16  9  4  1  0
```

```
void reverse(int src[], int dest[], int nelems)
{
    int spos = nelems - 1, dpos = 0;

    while (dpos < nelems)
        dest[dpos++] = src[spos--];
}
```

Arrays as parameters, continued

Problem: Write **int count_equal(int value, int a[], int nelems)**, that returns the number of elements in **a** that are equal to **value**.

Usage:

```
int a1[] = {2, 3, 1, 5, 4, 1, 1, 4};
int n = sizeof(a1) / sizeof(a1[0]);
iprint(count_equal(1, a1, n));
iprint(count_equal(5, a1, n));
```

Output:

```
count_equal(1, a1, n) is 3
count_equal(5, a1, n) is 1
```

Solution:

```
int count_equal(int value, int a[], int nelems) // arraypar3.c
{
   int count = 0;
   for (int i = 0; i < nelems; i++)
      if (a[i] == value)
         count++;
   return count;
}
```

Pitfall: Using **sizeof** with an array parameter

This **Java** method calculates the sum of the integers in an array:

```
static int sum(int a[ ])
{
    int sum = 0;
    for (int i = 0; i < a.length; i++)  // Java code!!
        sum += a[i];
    return sum;
}
```

Here is an attempt at a C version, but **IT DOES NOT WORK!!**

```
int sum(int a[])
{
    int sum = 0;
    for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++)
        sum += a[i];
    return sum;
}
intmain() // arraypar2.c
{
    int a[] = {10, 20, 30};
    printf("sum = %d\n", sum(a));  // Output: sum = 30 (!)
}
```

Pitfall: Using **sizeof** with an array parameter, continued

Let's investigate:

```
int sum(int a[])
{
    printf("sizeof(a) = %zu, sizeof(a[0]) = %zu\n",
        sizeof(a), sizeof(a[0]));
    int sum = 0;
    for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++)
        sum += a[i];
    return sum;
}
```

Call:

```
int a[] = {10, 20, 30};
printf("sum = %d\n", sum(a));
```

Output:

```
sizeof(a) = 8, sizeof(a[0]) = 4
sum = 30
```

What's going on?

Sidebar: `a` and `&a[0]`

Given an array `a`,

```
int a[5];
```

the expression

```
a
```

is equivalent to

```
&a[0]
```

Therefore, the call

```
sum(a)
```

is completely equivalent to

```
sum(&a[0])
```

<u>When an array is a function argument, what the function actually gets called with is the address of the first element of the array!</u>

Sidebar: a and &a[0], continued

At hand:

The calls **sum(a)** and **sum(&a[0])** are completely equivalent. In both cases what actually gets passed to **sum** is the address of the first element of **a**.

Here's **sum** again:

```
int sum(int a[])
{
    printf("sizeof(a) = %%zu, sizeof(a[0]) = %zu\n", sizeof(a),
sizeof(a[0]));
    ...
}
```

Output:

```
sizeof(a) = 8, sizeof(a[0]) = 4
```

Given **sum(int a[])**, and the equivalence of **a** and **&a[0]**,
sizeof(a) is equivalent to **sizeof(&a[0])**

The size of an address with **gcc** defaults on lectura is 8 bytes, so **sizeof(a)** is 8!

Pitfall: Using **sizeof** with an array parameter, continued

Here's the faulty routine again:

```
int sum(int a[])
{
    int sum = 0;
    for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++)
        sum += a[i];
    return sum;
}
```

sizeof(a), for <u>any</u> array parameter **a** of any type and size is 8!
(with **gcc** as we're using it)

NEVER USE **sizeof(a)** TO COMPUTE THE LENGTH OF AN ARRAY PASSED AS A FUNCTION PARAMETER.

Run-time array length information

In Java, ***array*.length** works because the array's internal data structure (usually) includes a length field. If we ran the JVM executable with **gdb**, we could examine each array and find its length.

We can say, "Java maintains run-time array length information."

C does not maintain run-time array length information (with the exception of variable length arrays—VLAs.)

Imagine one million two-element **char** arrays. How many bytes would be required to store those arrays?

Two million bytes.

Imagine that each array had run-time length information: a four-byte **int**. How many bytes would be required to hold the length information for those one million arrays?

Four million bytes.
(Twice as much as the data itself—200% overhead!)

Run-time array length information, continued

Let's compile this C function without -g:

```
% cat arraylen1.c
void f()
{
   int a[17],
       b[34],
       c[1000],
       d[888];
}
% \gcc -S arraylen1.c
```

The generated assembly code is on the right. There's no 17, 34, 1000, or 888 but there is some evidence of the arrays. What is it?

The instruction **subq $7656, %rsp** makes space on the stack for 7656 bytes:

sizeof(a)+sizeof(b)+sizeof(c)+sizeof(d)

(Well, almost...that sum is 7756!)

```
% cat arraylen1.s
.file   "arraylen1.c"
        .text
        .globl  f
        .type   f, @function
f:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $7656, %rsp
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   f, .-f
        .ident  "GCC: ... 4.6.3"
        .section        .note.GNU-stack...
```

Run-time array length information, continued

Let's see what **dstack** shows.

```
int main()
{
    int a[17], b[34], c[1000], d[888];

    paddr(&a[0]); paddr(&a[16]);
    paddr(&b[0]); paddr(&b[33]);
    ...
}
```

```
               |------------|
140734228016576 |   a[16]    |
               |------------|
                 <15 words>
               |------------|
140734228016512 |    a[0]    |
               |------------|
140734228016508 |    ...     |
               |------------|
140734228016504 |    ...     |
               |------------|
140734228016500 |   b[33]    |
               |------------|
                 <32 words>
               |------------|
140734228016368 |    b[0]    |
               |------------|
140734228016364 |   d[887]   |
               |------------|
                <886 words>
               |------------|
140734228012816 |    d[0]    |
               |------------|
140734228012812 |   c[999]   |
               |------------|
                <998 words>
               |------------|
140734228008816 |    c[0]    |
               |------------|
```

When a C program is being compiled, the compiler determines the size and location of all memory objects but the only trace of that left at run-time are constants that are pieces of machine instructions.

C arrays are an example of "What you write is what you get." If you say **int vals[10]**, space is consumed for ten **int**s and nothing more.

Run-time array length information, continued

Here again is our faulty **sum** function:

```
int sum(int a[])
{
   int sum = 0;
   for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++)
      sum += a[i];
   return sum;
}
```

Unless we want to introduce a sentinel value to signal the end of the data, our only choice is to pass an element count:

```
int sum(int a[], int nelems)
{
   int sum = 0;
   for (int i = 0; i < nelems; i++)
      sum += a[i];
   return sum;
}
```

An element count is not a length(!)

Passing an element count does provide some flexibility:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  //arraypar2a.c

for (int i = 1; i <= sizeof(a)/sizeof(a[0]); i++)
    printf("sum of first %d = %d\n", i, sum(a, i));
```

Output:

```
sum of first 1 = 1
sum of first 2 = 3
sum of first 3 = 6
sum of first 4 = 10
sum of first 5 = 15
...
```

Let's compute sums working backwards:

```
int alen = sizeof(a)/sizeof(a[0]);
for (int i = alen - 1; i >= 0; i--)
    printf("sum of last %d = %d\n",
                alen - i, sum(&a[i], alen - i));
```

Output:

```
sum of last 1 = 10
sum of last 2 = 19
sum of last 3 = 27
sum of last 4 = 34
sum of last 5 = 40
...
```

Because a C array parameter is nothing more than the address of the "first" element, a function can operate on a sequence of elements in the middle of an array! (But, we must be careful to stay in bounds.)

Is the loop above "lucky", with an off-by-one error?

You can't assign to an array variable!

Unlike Java, an array variable cannot be the target of assignment. Example:

```
% cat array9.c
int main()
{
   int a[5], b[5];
   a = b;
}

% gcc array9.c
array9.c:4:7: error: incompatible types when assigning to type
'int[5]' from type 'int *'
```

How does the compiler view `a = b`?

```
&a[0] = &b[0];
```

We'll later see how to achieve a similar effect using pointers.

Character strings

Character string basics

There is no character string type in C.

However, there is a well-established convention: a character string is represented by a zero-terminated sequence of **char** values.

One way to create a "string" is by initializing a **char** array with a sequence of values, being sure the last value is zero.

```
char s[ ] = { 'H', 'e', 'l', 'l', 'o', '!', '\n', 0 };
```

Here's a loop that prints the characters in **s**:

```
for (int i = 0; s[i] != 0; i++)
    printf("s[%d] = '%c' (%d)\n", i, s[i], s[i]);
```

Output:

```
s[0] = 'H' (72)
s[1] = 'e' (101)
s[2] = 'l' (108)
s[3] = 'l' (108)
s[4] = 'o' (111)
s[5] = '!' (33)
s[6] = '
' (10)
```

What's **s[7]**?

Character string basics, continued

For reference:

```
char s[ ] =  { 'H', 'e', 'l', 'l', 'o', '!', '\n', 0 };
```

Here are two completely equivalent ways to initialize **s**:

```
char s[ ] = { 72, 101, 108, 108, 111, 33, 10, 0};

char s[ ] = { 72, 'e', 108, 25*4+8, 0x6F, '\041', sizeof("abcd")*2, 0};
```

Is the following valid?

```
char s2[ ] =  { 'H', 'e', 'l', 'l', 'o', '!', '\n', 0, 'x', 0 };
```

Character string basics, continued

Problem: Write a routine **void pstring(char s[])** that prints the chararacters in the "string" **s** on standard output. Example:

```
void pstring(char s[]);
int main() // string1.c
{
   char s1[ ] = { 'H', 'e', 'l', 'l', 'o', '!', '\n', '\0' };

   char s2[ ] = { 72, 101, 108, 108, 111, 33, 10, 0};

   char s3[ ] = { 'H', 'e', 'l', 'l', 'o', '!', 10, 0, 'x', 'y', 'z', 0 };

   pstring(s1);
   pstring(s2);
   pstring(&s3[1]);
}
```

Output:

```
Hello!
Hello!
ello!
```

Character string basics, continued

One solution:

```
void pstring(char s[])
{
   for (int i = 0; s[i] != 0; i++)
      putchar(s[i]);
}
```

A more idiomatic solution:

```
void pstring(char s[])
{
   for (int i = 0; s[i]; i++)
      putchar(s[i]);
}
```

Which do you prefer?

Exercise: The library routine **puts** is similar to **pstring**. Do **man puts** and see how **puts** differs from **pstring**. Ignore the argument type of **puts** and write a simple test of **puts**. As the SYNOPSIS shows, you'll need to include **stdio.h**.

Character string basics, continued

Problem: Write a routine **int length(char s[])** that returns the length of string **s**.

```
int length(char s[]);
int main() // string1a.c
{
    char line[] = { 'a', 'b', 'c', 0};

    for (int i = 0; i < sizeof(line); i++)
        printf("length(&line[%d]) = %d\n",
          i, length(&line[i]));
}
```

Output:

```
length(&line[0]) = 3
length(&line[1]) = 2
length(&line[2]) = 1
```

A solution:

```
int length(char s[])
{
    int i = 0;
    while (s[i])
        i++;

    return i;
}
```

strlen in the library is very similar to **length** above. Use **man** to see how it differs.

Character string basics, continued

Consider this code:

```
char line[] = { 'a', 0, 'b', 'c', 0, 'd', 'e', 'f', 0 };

for (int i = 0; i <= sizeof(line); i++)
    printf("length(&line[%d]) = %d\n", i, length(&line[i]));
```

Output:

```
length(&line[0]) = 1
length(&line[1]) = 0
length(&line[2]) = 2
length(&line[3]) = 1
length(&line[4]) = 0
length(&line[5]) = 3
length(&line[6]) = 2
length(&line[7]) = 1
length(&line[8]) = 0
length(&line[9]) = 2
```

How many strings are in **line**?

Any bugs in the loop above?

Character string basics, continued

Problem: Write a function **int pos(char s[], char c)** that returns the position of the first occurrence of **c** in **s**. **pos** returns **-1** if **c** is not found in **s**.

```
              0  1  2  3  4  5  6   7   8
char s[] = { 't', 'e', 's', 't', 'i', 'n', 'g', '\n', 0};

iprint(pos(s, 's'));
iprint(pos(s, 'x'));
iprint(pos(s, 1101 / 10));
iprint(pos(&s[1], 't'));
```

Output:

```
pos(s, 's') is 2
pos(s, 'x') is -1
pos(s, 1101 / 10) is 5
pos(&s[1], 't') is 2
```

Solution:

```
int pos(char s[], char c)
{
    for (int i = 0; s[i]; i++)
        if (s[i] == c)
            return i;

    return -1;
}
```

Note: There's no direct C library equivalent for **pos**. **strchr(s, c)** searches for the character **c** in the string but it returns the address of the character instead of its "index".

Sidebar: `0` vs. `'\0'` vs. `'0'`

Here are three constants that are commonly confused:

```
0
'\0'
'0'
```

One by one:

What's `0`?

`0` is an `int` constant.

What's `'\0'`?

`'\0'` is an octal character constant. The surrounding single quotes designate a character constant. Inside a character constant, a backslash followed by one, two or three octal digits is the specification of a character by its octal code.

Note that `'\0' == 0`, just like `'\101' == 65`

`'\0'` and `0` are two ways to specify the same numeric value: zero! Here are two more ways to specify zero: `0x0` and `'\x00'`.

Sidebar: `0` vs. `'\0'` vs. `'0'`, continued

What's `'0'`?

`'0'` is a character constant that represents the 49th character in the ASCII character set, that has code 48. Here is the graphic for it: `0`

Note that `'0' == '\060'` and `'\060' == 48`

Remember that the type of a character constant is `int`, not `char`.

Another value in the swirl for some is `"0"`. We'll focus on it soon!

String literals

Needless to say, it's tedious to initialize an array of **char** values like this:

```
char s[] = { 'a', 'b', 'c', 0 };  // Equivalent: {97, 98, 99, 0}
```

As a convenience, C allows a string literal to be used to initialize a **char** array.

Here is a completely equivalent initialization:

```
char s[] = "abc";
```

Note that there is no trailing zero. The convention of zero-termination of strings is so strong that a zero-valued byte is added to every C string literal.

The initialization

```
char s1[] = "abc\0";
```

is equivalent to this:

```
char s1[] = { 'a', 'b', 'c', 0, 0 };
```

Speculate: What does **char s2[] = ""** create?

```
char s2[] = { 0 };
```

String literals, continued

We've seen **printf** calls like this:

```
printf("%d\n", i);
```

The end result of the above line is very similar to this:

```
char _tmp273[] = "%d\n";
printf(_tmp273, i);  // recall: just like printf(&_tmp273[0], i);
```

(We're imagining that **_tmp273** is an internal, compiler-generated identifier.)

The above in turn is identical to this:

```
char _tmp273[] = { '%', 'd', '\n', 0};
printf(_tmp273, i);
```

String literals, continued

What is `sizeof("abc")`?

    4
    { 'a', 'b', 'c', 0 }

What is `sizeof("\n\n")`?

    3
    { '\n', '\n', 0 }

What is `sizeof("")`?

    1
    { 0 }

What is `sizeof("\0")`?

    2
    { 0, 0 }

What is `sizeof("\0/")`?

    3
    { 0, '/', 0 }

String literals, continued

Let's combine literals with some of the string functions we've written:

```
pstring("Hello!\n");

pstring(&"Hello!\n"[5]);

iprint(pos("testing", 's'));

iprint(length(&"string"[2]));

iprint(length("abc\0def"));
```

Output:

```
Hello!

!

pos("testing", 's') is 2

length(&"string"[2]) is 4

length("abc\0def") is 3
```

pstring vs. printf("%s", ...)

For reference:

```
void pstring(char s[])
{
   for (int i = 0; s[i]; i++)
      putchar(s[i]);
}
```

In essence, the argument of **pstring** is assumed to be the starting address of a sequence of **char** values to print. **pstring** calls **putchar()** with each value in turn until a zero is reached.

The **%s** format specifier of **printf** does essentially the same work as **pstring**: given a starting address it outputs **char** values as characters until a zero is reached. Given

```
char greeting[ ] = "Hello\n";
```

then the same output is produced by both of these statements:

```
pstring(greeting);
printf("%s", greeting);
```

Sidebar: *Syntactic sugar*

Wikipedia says,

> "*Syntactic sugar* is syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer."

In C, **char s[] = "abc"** is syntactic sugar. It means exactly the same thing as **char s[] = { 'a', 'b', 'c', 0 }** but it's much more convenient.

Are character constants an example of syntactic sugar? Wouldn't **65** do for '**A**'?

Are **while**, **do-while**, and/or **for** examples of syntactic sugar? How about **goto**?

Given **int a[5]**, is the **a** and **&a[0]** equivalence an example of syntactic sugar?

What are examples of syntactic sugar in Java?

Wikipedia says the term "syntactic sugar" was coined by Peter J. Landin in 1964.

"Syntactic sugar causes cancer of the semi-colons."—Alan Perlis

Strings and "lucky" programs

Here is a "lucky" program:

```
int main() // luckystring1.c
{
   char s[10];

   printf("s = >%s<\n", s);
}
```

Output:

```
% (gcc luckystring1.c && a.out) | cat -v
s = ><
```

What's lucky (or not) about the program?

s is uninitialized!

What can be said about **s[0]**, based on the output above?

Since no characters appear between **>** *and* **<**, ***s[0]*** *must be zero.*

How about **s[1]**?

Nothing is apparent about ***s[1]***.

Strings and "lucky" programs, continued

A revision:

```
int main() // luckystring1.c
{
    char s[25];  // was char s[10];
    printf("s = >%s<\n", s);
}
```

Output:

```
% (gcc luckystring1.c && a.out) | cat -A
s = >M-^?M-2M-p<$

% (gcc luckystring1.c && a.out) | od -td1
0000000  115   32   61   32   62   -1  -78  -16   60   10
0000012
```

What's happening?

<u>When we change the size of **s**, its starting point also changes.</u> If a zero byte happens to be where **s[0]** is, we get just **s = ><**. If non-zero byte happens to be where **s[0]** is, we get one or more random characters output between **>** and **<**.

Strings and "lucky" programs, continued

This program prints the values for the first few bytes of **s**. But, what's **SIZE**?

```
int main() // luckydump1.c
{
   char s[SIZE];
   for (int i = 0; i < 10; i++)
      printf("%5d", s[i]);
   printf("\n");
   printf(">%s<\n", s);
}
```

Using -D*MACRO*=*VALUE* on the command line has the same effect as if we had the following **#define** as the first line of **luckydump1.c**:

```
#define MACRO  VALUE
```

Example:

```
% gcc -DSIZE=10 luckydump1.c  && a.out | cat -v
    0    0    0    0    0    0    0    0  -48    4
><
```

Strings and "lucky" programs, continued

Let's do -DSIZE=10 again, and a few more:

```
% gcc -DSIZE=10 luckydump1.c  && a.out | cat -v
    0    0    0    0    0    0    0    0  -48    4
><
% gcc -DSIZE=5 luckydump1.c  && a.out | cat -v
  -64   -6   18  -33   -4  127    0    0    0   31
>M-@M-z^RM-_M-|^?<
% gcc -DSIZE=1000 luckydump1.c  && a.out | cat -v
    0    0    0    0    1    0    0    0 -126    8
><
% gcc -DSIZE=1000000 luckydump1.c  && a.out | cat -v
    0    0    0    0    0    0    0    0    0    0
><
```

Let's make a bash loop!

```
% for size in $(seq 10 100 50000)
> do
> echo $size...
> gcc -DSIZE=$size luckydump1.c  && a.out | cat -v
> done
```

Strings and "lucky" programs, continued

It is essential to understand the following:

- A function **`f(char s[])`** has no way to know whether the memory referenced by the parameter **`s`** is valid data.
- **`f`** has no choice but to assume that **`s`** references valid data and proceed.
- A string-processing function **`f`** will typically process characters until it reaches a character whose value is zero.

Strings and "lucky" programs, continued

Here is a routine that creates a string in **s** that is **n** copies of a character **c**.

```
void fill(char s[], char c, int n) // luckystring3.c
{
   for (int i = 0; i < n; i++)
      s[i] = c;
}
```

Usage:

```
char buf[30];

fill(buf, 'a', 3);
printf("%s\n", buf);

fill(buf, 'x', 10);
printf("%s\n", buf);

fill(buf, 'e', 5);
printf("%s\n", buf);
```

Execution:

```
% a.out | cat -v
aaa
xxxxxxxxxx@
eeeeexxxxx@
```

What's happening?

fill isn't following the **n** copies of **c** with a zero-valued byte.

s[3] happened to be zero.

s[10] happened to hold '@', and **s[11]** happened to be zero.

The five **e**'s overwrote the first five **x**'s but nothing more.

Strings and "lucky" programs, continued

Let's fix **fill** by "capping off" the result in **s** with a zero:

```
void fill(char s[], char c, int n) // luckystring3a.c
{
   for (int i = 0; i < n; i++)
      s[i] = c;
   s[n] = 0;    // Added
}
```

Usage: (unchanged)

```
char buf[30];

fill(buf, 'a', 3);
printf("%s\n", buf);

fill(buf, 'x', 10);
printf("%s\n", buf);

fill(buf, 'e', 5);
printf("%s\n", buf);
```

Execution:

```
% a.out | cat -v
aaa
xxxxxxxxxx
eeeee
```

Working with strings

Consider the problem of "assignment" with strings. This does not work:

```
char a[] = "pickles", b[10];

b = a;  // Compilation error! (&b[0] = &a[0])
```

The workable alternative is a routine: `copy(char to[], char from[])`

Usage:

```
char a[] = "pickles", b[10];

copy(b, a);  // Sort of like 'b = a'

printf("b = >%s<\n", b);  // Output: b = >pickles<
```

The right-to-left "movement" mimics the assignment operator.

Problem: Write it!

Working with strings, continued

For reference:

```
char a[] = "pickles", b[10];

copy(b, a); // Sort of like 'b = a'

printf("b = >%s<\n", b);  // Output: b = >pickles<
```

An implementation of copy:

```
void copy(char to[], char from[])
{
   int i = 0;
   while ((to[i] = from[i])) // extra parens to avoid warning
      i++;
}
```

Do we need **to[i] = 0**, too?

Working with strings, continued

What's the output of the following code? (from **strexcopy1.c**)

```
char a[] = "pickles", b[10];

copy(b, "noodles");
printf("b = >%s<\n", b);
        // Output: b = >noodles<

copy(b, "chicken noodle soup");
printf("b = >%s<\n", b);
        // Output: b = >chicken noodle soup<

printf("a = >%s<\n", a);
        // Output: a = >oup<
```

Why?

b is too small for "chicken noodle soup"

What can we say about the relative position of **a** and **b**?

Since b's overrun overwrote a, a must follow b.

Working with strings, continued

At hand:

```
void copy(char to[], char from[])
{
   int i = 0;
   while ((to[i] = from[i]))
      i++;
}
int main()
{
   char b[10];
   copy(b, "chicken noodle soup");
}
```

Problem: Modify **copy** so that it detects a too-small **to** array, like above.

Impossible!

Here's what we might see in some call to **copy** when we look at values in **to**:

43 167 207 112 45 13 24 3 34 187 88 78 92 133 235

What's the last byte in **to**?

How about in this one?

97 98 99 0 0 0 0 48 49 50 51 0 1 0 1 216 70 60 198

Working with strings, continued

Consider the following routine:

```
int rln(char s[])
{
    int c, spos = 0;

    while ((c = getchar()) != EOF) {
        if (c == '\n')
            break;
        s[spos++] = c;
        }

    if (c == EOF && spos == 0)
        return EOF;
    s[spos] = 0;
    return spos;
}
// strexrln.c
// In h/strfuncs.h, too.
```

What high-level operation does it perform?
Reads a newline-terminated line into **s**.

What does a call look like?

```
char line[100];
rln(line);
```

What assumptions does **rln** make?
rln assumes that **line** is big enough!

How could **rln** be exploited by an attacker?
A carefully constructed input string could overflow the caller's "buffer" and overwrite data in the caller's "stack frame". Ultimately, an attacker could "execute arbitrary code".

Code using **rln** is subject to a *buffer overflow (overrun) attack*.

Working with strings, continued

Problem: Write a routine **concat(char to[], char new[])** that appends the contents of the string **new** to the string **to**. Example:

```
char s[100];

copy(s, "just");
concat(s, " testing");
concat(s, " this");
printf("s = >%s<\n", s); // Output:  s = >just testing this<
```

Solution:

```
void concat(char to[], char from[])
{
   int topos = 0, frompos = 0;

   while ((to[topos]))
      topos++;

   while ((to[topos++] = from[frompos++]))
      ;
}
```

Would it be better to write **concat** with **length** and **copy** instead? (Do it!)

Working with strings, continued

Consider the following program. Note that **readline()** is **rln()** from slide 251.

```
int main() // strex3.c
{
    char buf[1000000], line[1000];

    while (readline(line) != EOF) {
        concat(buf, ">");
        concat(buf, line);
        concat(buf, "<\n");
        }

    printf("%s", buf);
}
```

In broad terms, what does the program do?

What assumptions does it make?

No line is longer than 999 bytes.

Full input plus wrapping '>'s and '<'s is less than a million characters.

What vulnerabilities does it have?

See assumptions!

Are there any bugs?

What is the initial value of **buf[0]**?

What's a three-character fix?

char buf[1000000] =\{\}

Characterize the running time using *O(n)* notation.

C library counterparts for examples

The functions **pstring**, **length**, **copy**, **concat**, and **rln**/**readline** presented in the preceding slides are simplified versions of C library functions.

Slides	C library	Notes
pstring	**puts**	
length	**strlen**	**strlen** returns a value of type **size_t**, which is a "**typedef**" for an **unsigned long**, but for our purposes **strlen**'s result can be safely assigned to an **int**. Use a **%zu** format when a **strlen** call is an argument to **printf**: **printf("%zu\n", strlen("x"))**.
copy	**strcpy**	**strcpy** returns the address of the destination string instead of **void**. See also **strncpy**.
concat	**strcat**	**strcat** returns the address of the destination string instead of **void**. See also **strncat**.
readline (a.k.a. **rln**)	**gets**	**gets** returns its argument when successful, but **NULL** (a "null pointer") at **EOF**.

To use the functions from the slides, **#include "strfuncs.h"**. Prototypes for the **str*** functions are in **<string.h>**. Prototypes for **gets** and **puts** are in **<stdio.h>**.

Pointers

Pointer basics

Given 'int i;' then &i produces the address of i.

The type of &i is said to be pointer to int, or int *, or "int star".

A pointer variable can hold an address. A pointer variable is declared by preceding the variable's name with an asterisk:

```
int *p;
```

The type of p is "pointer to int".

A value with the type "pointer to int" can be assigned to p:

```
p = &i;
```

We might now say "p points to i", or "p references i", or "p holds the address of i". (Three ways to say the same thing.)

Note that i is not initialized. It's ok for a pointer to reference uninitialized memory.

Pointer basics, continued

Like any other variable, the value of a pointer variable can be changed.

Let's work through the following code with pictures.

```
int i, j, a[2];
int *p;

p = &i;

p = &j;

p = &a[0];

p = &a[1];

p = a;

p = &a[100];
```

Pointer basics, continued

Given **int** ***p**,

the value of the expression ***p** is the value of the **int** that's referenced by **p**.

Example:

```
int i = 7, j, k;
int *p;

p = &i;

j = *p;             // assigns 7 to j

k = *p +  *p; // assigns 14 to k
```

Pointer basics, continued

If `*p` appears on the left hand side of an assignment, the `int` referenced by `p` is the target of the assignment.

```
int i = 7, j, k;
int *p;

p = &i;

*p = 5;

p = &j;

*p = 10;

*p *= 2;

int *q = &i;

*q = *p;
```

Sidebar: Declaration mimics use

Here's an important observation about C:

Declaration mimics use

The declaration

```
int i;
```

can be thought of as saying "If you see the expression **i**, it is an **int**."

Similarly,

```
int a[5];
```

can be thought of as saying "If you see the expression **a[n]**, it is an **int**."

Likewise,

```
int *p;
```

can be thought of as saying "If you see the expression ***p**, it is an **int**."

What does **int *a, b, *c, d;** declare?

- **a** is a "pointer to **int**".
- **b** is an **int**.
- **c** is "pointer to **int**".
- **d** is an **int**.

How about **int* p1, p2, p3**?

p1 is a pointer; **p2** and **p3** are **int**s.
(whitespace makes no difference)
How could you check it?

Sidebar: Declaration mimics use, continued

All three can be declared at once:

```
int i, *p, a[5];
```

Again: **i** is an int, ***p** is an int, **a[n]** is an **int**.

Imagine that "declaration mimics use" wasn't used. What's an alternative syntax that might have been chosen?

```
declare i: int, p: address of int, a: array of 5 int;
```

What's a downside to this more English-like form?

- We'd need to know two forms instead of one.
- The compiler would definitely need entirely separate code to parse it.
- The standard would need to fully describe this alternate system.

"Declaration mimics use" gives C a smaller "mental footprint": if you know how you want to use something in C, you know how to declare it. (Mostly...)

Pointer basics, continued

Here's a loop that fills an array a with squares of 0 through 4:

```
int a[5];

for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    int *p = &a[i];   // Equivalent:  int *p;  p = &a[i];
    *p = i * i;
    }
```

Here's a loop that sums the values in the array a:

```
int sum = 0;

for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    int *p = &a[i];
    sum += *p;
    }
```

Note that the above is a non-idiomatic mix of pointers and array subscripting; it's just illustrating some mechanics.

Pointer basics, continued

The value of a pointer variable may be assigned to another variable of the same pointer type.

Any number of pointers may reference the same object in memory.

```
int i, j;
int *p1, *p2;

p1 = &i;
p2 = p1;

*p1 = 15;

j = *p2;

printf("i = %d, j = %d\n", i, j);
                    // Output: i = 15, j = 15
```

Pointers as arguments

A pointer can be passed to a function as an argument:

```
int twice(int *p);
int main() // twice.c
{
   int i = 7;
   int *p = &i;
   paddr(p);
   i = twice(p);
   iprint(i);
}
int twice(int *ip)
{
   paddr(ip);
   return *ip * 2;
}
```

Output:

```
p = 140734133231676
ip = 140734133231676
i is 14
```

Pointers as arguments, continued

How does this variant, **twice2**, differ from **twice**?

```
void twice2(int *p);

int main()  //twice2.c
{
    int i = 7;
    int *p = &i;

    twice2(p);
    iprint(i);
}

void twice2(int *ip)
{
    *ip = *ip * 2;
}
```

Output:

```
i is 14
```

int twice(int *ip), on the previous slide, takes a pointer to an **int** and returns twice the value of the integer.

void twice2(int *ip) takes a pointer to an **int** and doubles its value in place. Note that **twice2** does not return a value!

Does this example conflict with the claim that C uses call-by-value?

No. The value of **p** is being passed to **twice2**, but that value is the address of the variable **i**.

Here's another view:

twice2(A) doubles the **int** at address **A**.

Pointers as arguments, continued

Problem: Write a swap function. Example: (**swap0.c**)

```
int i = 7, j = 11;

swap(&i, &j);

printf("i = %d, j = %d\n", i, j);   // Output: i = 11, j = 7
```

Solution:

```
void swap(int *ap, int *bp)
{
   int t = *ap;
   *ap = *bp;
   *bp = t;
}
```

Can we write a Java analog for **swap**?

No.

Exercise: Draw a diagram of memory and work your way through a call to **swap**.

Pointers as arguments, continued

At hand:

```
void swap(int *ap, int *bp)
{
   int t = *ap;
   *ap = *bp;
   *bp = t;
}
```

What is the result of the following code?

```
int i = 7, j = 11, a[2];

swap(&i, &a[0]);
swap(&j, &a[1]);

printf("a[0] = %d, a[1] = %d\n", a[0], a[1]);
printf("i = %d, j = %d\n", i, j);
```

Output:

```
a[0] = 7, a[1] = 11
i = -365601136, j = 32764
```

Pointers as arguments, continued

Here's a routine that tallies the number of even and odd values in an array:

```
void tally_eo(int a[], int size, int *evenp, int *oddp)  // tally_eo.c
{
    int evens = 0, odds = 0;

    for (int i = 0; i < size; i++)
        if (a[i] % 2 == 0)
            evens += 1;
        else
            odds += 1;

    *evenp = evens;
    *oddp = odds;
}
```

Usage:

```
int a[] = { 1, 3, 4, 5, 7}, e, o;
tally_eo(a, sizeof(a)/sizeof(a[0]), &e, &o);
printf("evens = %d, odds = %d\n", e, o);
        // Output: evens = 1, odds = 4
```

Does **tally_eo** need the locals **evens** and **odds**?

How would **tally_eo** be formulated in Java?

Exercise: What does the following statement do?

```
tally_eo(&a[2], sizeof(a)/sizeof(a[0]),
        &a[1], a);
```

Pointers as return values

Here is a function that returns a pointer to the largest value in an array of one or more values:

```
int *maxval(int vals[], int nelems) // maxval.c
{
    int *maxp = &vals[0];
    for (int i = 1; i < nelems; i++)
        if (vals[i] > *maxp)
            maxp = &vals[i];

    return maxp;
}
```

Usage:

```
int a[] = {10, 17, 3, 18, 27};
printf("max is %d\n", *maxval(a, 5));  // Output: max is 27
```

Declaration mimics use: The type of the expression ***maxval(...)** is **int**.

Pointers as return values, continued

Another use of **maxval**:

```
int a[] = {10, 17, 3, 18, 27};

for (int n = 1; n <= sizeof(a)/sizeof(a[0]); n++) {
   int *mp = maxval(a, n);
   printf("max in first %d, at %p, is %d\n", n, mp, *mp);
   }
```

Output:

```
max in first 1, at 0x7ffcfa426fb0, is 10
max in first 2, at 0x7ffcfa426fb4, is 17
max in first 3, at 0x7ffcfa426fb4, is 17
max in first 4, at 0x7ffcfa426fbc, is 18
max in first 5, at 0x7ffcfa426fc0, is 27
```

Which is more useful, the index of an element in an array or the address of the element?

One fact: An address is sufficient to access a value but an index needs to be paired with an array.

Another fact: Sometimes we want to know the position of a value in an array.

Pointers and types

It is important to understand that **int *p** doesn't simply declare **p** to be a pointer. It declares **p** to be a pointer to an **int**. This creates a pair of requirements for **p**:

- The expression ***p** can only be used in contexts where an **int** is permitted.
- The type of a value assigned to **p** must be "pointer to **int**".

Given **char c**, the type of **&c** is "pointer to **char**". Assigning pointer values of incompatible types generates a warning (but not an error). Example:

```
char c;
int *p;

p = &c; // warning:  assignment from incompatible pointer type

int i;
char *p2;
p2 = &i; // warning:  assignment from incompatible pointer type
```

Pointers and types

On lectura with **gcc**, pointers are simply 64-bit integers but attempting to use a pointer as an integer (or vice-versa) generates a warning:

```
int *p;  long i;

i = &i;        // "assignment makes integer from pointer without a cast"

p = 1000;   // "assignment makes pointer from integer without a cast"
```

As the warnings suggest, a *cast* silences the warnings:

```
i = (long)&i;                    // No warning

p = (int*)0x7ffcfa426fb0; // No warning
```

Casts in C are essentially the same as casts in Java: A type name enclosed in parentheses preceding an expression indicates a desired type for the resulting value.

Our **paddr** macro uses a cast to treat a pointer as a **long unsigned**:

```
#define paddr(a) printf(#a " = %lu\n", (long unsigned)a)
```

Sidebar: Are pointer types really needed?

Speculate: If pointers are simply 64-bit integers, why not allow the following?

```
long i, p;
p = &i
*p = 7;        // (1)
```

That is, do we really need pointer types? Why not just store addresses in **long**s?

Let's add more code to the picture:

```
char s[] = "abc";
p = &s[1];
*p = 'x';      // (2)
```

How many bytes should change on line (1)?

p points to the **long i**, so the eight bytes of **i** should be changed to hold **7**.

How about line (2)?

p points to a **char** in **s**, so one byte should be changed to hold '**x**'.

Is there an inconsistency between (1) and (2)?

Sidebar: Are pointer types really needed?

At hand:

```
long i, p;
p = &i
*p = 7;        // should change eight bytes

char s[] = "abc";
p = &s[1];
*p = 'x';      // should change one byte
```

The problem: without knowing the size of a pointed-to memory object, the compiler doesn't know how many bytes to move!

A declaration like **int *p2** says that **p2** is a pointer that points to an **int**.

The expression ***p2 = 5** says to change **sizeof(int)** bytes at the address held in **p2**.

Bottom line: In addition to detecting inconsistencies at compile-time, pointer type information implies the size of memory objects.

Sidebar: Are pointer types really needed?

How are data value sizes specified in assembly language?

- Some machines have specific instructions to move 1, 2, 4, or 8 bytes.
- Some machines have instructions with width fields.
- And more...

Recall this from the history slides:

> Thompson starts to write a FORTRAN compiler but changes direction and creates a language called "B", a cut-down version of BCPL. ("Basic Combined Programming Language")
>
> B was interpreted and was oriented towards <u>untyped word-sized objects</u>. Dennis Ritchie adds types and writes a compiler. Ritchie initially calls it NB for New B, but renames it C. (1972)

With "untyped word-sized objects", all pointers reference objects of the same size: a word!

Null pointers

The constant 0 (zero) can be used as a pointer value of any type:

```
int *p1 = 0;        // No warning...
char *p2 = 0;       // No warning...
```

A zero that is used as a pointer is called a *null pointer*. Given the initializations above we can say that `p1` and `p2` are null pointers.

The C11 standard requires that `<stddef.h>` define a macro `NULL` that expands to a null pointer constant. With `gcc` on lectura, the macro `NULL` expands to `((void *)0)`.

Here are three completely equivalent ways to see if a `p` is null pointer:

```
if (p == NULL) ...
if (p == 0) ...
if (!p) ...
```

There are some detailed rules regarding the internal representation of null pointers. One example, from old notes: `p1 = 0` and `p2 = (int*)(i - i)` might not produce the same value in `p1` and `p2`!

Sizes of pointers

It is common (but not guaranteed) that the sizes of all pointers are the same. Here are sizes produced with gcc on lectura:

```
sizeof(char *)        // 8
sizeof(int *)         // 8
sizeof(double *)      // 8
sizeof(long long *)   // 8
```

Given

```
char *cp;
int *ip;
long *lp;
```

What are these?

```
sizeof(cp)
    8
sizeof(*cp)
    1
sizeof(ip)
    8
sizeof(*ip)
    4
sizeof(lp)
    8
sizeof(*lp)
    8
```

L-values and R-values

L-values and R-values

A simple understanding of the unary * operator is that in an expression, `*p` produces a value but as the target of an assignment it specifies a destination in memory. A deeper understanding is required to use C effectively.

An important concept in C is that of the "L-value":

An L-value is an expression that specifies an object in memory.

The simplest example of an L-value is a variable name:

Given a declaration such as `int x`, the expression `x` refers to an `int`-sized memory object (simply four contiguous bytes of memory) that is reserved to hold the value of `x` whenever `x` is "live".

Here is the general form of the unary ampersand operator:

&L-value

Assuming that `x` is in scope, an expression such as `&x` is permitted because `x` specifies an object in memory.

L-values and R-values, continued

For reference:

An L-value is an expression that specifies an object in memory.

Given **char s[5]**, is **s[0]** an L-value?

Yes. **s[0]** specifies the memory object that is the first element of **s**, a **char**.

Given **char s[5]**, is **s[100]** an L-value?

&s[100] will compile without error, proving that the compiler considers **s[100]** to be an L-value but run-time behavior is undefined if we use it.

An expression such as **x + y** is not an L-value: the standard does not guarantee that the result of the addition will ever have a memory object associated with it.

Because **x + y** is not an L-value, the expression **&(x + y)** is not valid.

Is **&100** valid?

No. There's no guarantee that there will ever be a **100** at any specific place in memory.

L-values and R-values, continued

For reference:

An L-value is an expression that specifies an object in memory.

Don't confuse L-values and addresses. An L-value specifies an object in memory. In turn, every object in memory has an address.

The "L" in "L-value" comes from "left". Here's an informal definition:

"An L-value is something that can appear on the left-hand side of an assignment."

Consider:

```
int x, y;
char s[5];
x = 3;
s[2] = 'z';

(x + y) = 5;  // error: lvalue required as left operand of assignment
10 = 20;      // ditto
```

L-values and R-values, continued

For reference:

An L-value is an expression that specifies an object in memory.

Consider this expression:

`x = y`

In C, like most languages, it means this:

"Fetch the value contained in ***`y`*** *and store it in x."*

One perspective the compiler has on **`x = y`** is this:

L-value = L-value

The two L-values are treated differently based on their context:

- The L-value on the left-hand side of the assignment names the destination of the assignment.
- The L-value on the right-hand side names the source of the value to be assigned.

L-values and R-values, continued

The value contained in a memory object is sometimes called the R-value.

Given `int x, y = 7`, the R-value of `y` is `7`. (Remember that `x` and `y` are L-values.)

An important rule:

If an L-value is provided but an R-value is needed, the value contained in the memory object named by the L-value is fetched.

For a simple assignment involving two variables, the compiler ultimately wants to arrive at this:

L-value = R-value

For `x = y` the goal is achieved by fetching the value contained in `y` and storing it in the memory designated for `x`.

L-values and R-values, continued

In general, if an expression contains L-values the computation is performed using the R-value of each L-value.

Constants such as **7**, **2.345**, and '**a**' have an R-value but no L-value.

Consider this expression:

```
a[i-j] = x + y * 2 - z / 3
```

What are the L-values in it?

a, **i**, **j**, **a[i-j]**, **x**, **y**, and **z**

Contrast:

In the BLISS programming language, the expression **x = y** means "Assign the <u>address</u> of **y** to **x**." (like **x = &y** in C).

If the value of **y** is to be assigned, the appropriate BLISS expression is this:

```
x = .y
```

(The unary dot operator specifies that the R-value of **y** is desired.)

L-values and R-values, continued

Earlier, this was said:

"*Given* `int *p` *the expression* `*p` *has the value of the* `int` *that is referenced by* `p`."

In fact, the result of `*p` is an L-value that specifies the object referenced by `p`.

Remember: An L-value is an expression that specifies an object in memory.

Consider this code:

```
int i, j = 7;
int *p1 = &i, *p2 = &j;
*p1 = *p2;
```

The compiler regards

```
*p1 = *p2
```

as

```
L-value = L-value
```

Because `*p1` is on the left-hand side of the assignment, the memory object it references is the target of the assignment. Because `*p2` is on the right, where an expression is required, the R-value of `*p2` is used.

L-values and R-values, continued

For reference:

```
int i, j = 7;
int *p1 = &i, *p2 = &j;

*p1 = *p2;
```

Because `*p1` and `*p2` are L-values, this code is valid:

```
int *p3, *p4;

p3 = &*p1;
p4 = &*p2;
```

Problem: Describe the values held in `p3` and `p4`.

L-values and R-values, continued

Here are two definitions from *The C Programming Language, 1st edition*:

&L-value

The result of the unary **&** operator is a pointer to the object referred to by the L-value.

If the type of the L-value is '**...**' then the type of the result is '**pointer to ...**'.

**expression*

The unary ***** operator means indirection: the expression must be a pointer, and the result is an L-value referring to the object to which the expression points.

If the type of the expression is '**pointer to ...**', the type of the result is '**...**'.

Given **int x**, what is the type of **&x**?

x is an L-value. The type of **x** is **int**. The type of **&x** is **pointer to int**.

Given **int *p**, what is the type of ***p?**

The type of **p** is **pointer to int**. The type of ***p** is **int**. ***p** is an L-value.

L-values and R-values, continued

For reference:

If the type of an **L-value** is '**...**' then the type of **&L-value** is '**pointer to ...**'.

If the type of **p** is '**pointer to ...**', the type of ***p** is '**...**'.

Let's work with these declarations:

```
int i = 7 , *p = &i;     // assume &i is 100
int a[5] = { 33 };       // assume &a[0] is 120
```

Expression	Type	Is it an L-value?	What is the R-value?
i	**int**	yes	**7**
&i	pointer to **int**	no	**100**
p	pointer to **int**	yes	**100**
***p**	**int**	yes	**7**
7	**int**	no	**7**
a[0]	**int**	yes	**33**
&a[0]	pointer to **int**	no	**120**
&*p	pointer to **int**	no	**100**

Pointer arithmetic

Pointers can participate in a limited set of arithmetic operations. One operation that's permitted is addition of a pointer and an integer.

Example: (ptrarith1.c)

```
char s[] = "abc";
char *p;
p = s;      // Like p = &s[0];

while (*p) { // Was *p != 0 -- Doh!
    printf("p = %lu, *p = %lu (%c)\n", p, *p, *p);
    p = p + 1;
    }
```

Output:

```
p = 140726812704416, *p = 97 (a)
p = 140726812704417, *p = 98 (b)
p = 140726812704418, *p = 99 (c)
```

What are alternatives for `p = p + 1`?

```
p += 1

p++

++p

p = 1 + p
```

Pointer arithmetic, continued

Here's another example of pointer arithmetic:

```
int a1[] = {13, 4, 109};

int *ip = a1;
for (int i = 1; i <= 3; i++) {
    printf("ip = %lu, *ip = %d\n", ip, *ip);
    ip++;
    }
```

Output:

```
ip = 140733814619776, *ip = 13
ip = 140733814619780, *ip = 4
ip = 140733814619784, *ip = 109
```

Is there anything surprising in that output?

How much does **ip++** add to **ip**?

4

140733814619776 + 1 == 140733814619780 (!?)

Why?

Pointer arithmetic, continued

An important rule:

If an integer is added to or subtracted from a pointer, the integer is "scaled" by the size of the pointed-to object.

In general, given **T *p;** then the value of

```
p + n
```

is

```
(int)p + n * sizeof(*p)    // Type of result: T*
```

Similarly, given **T *p**,

```
p - n
```

is

```
(int)p - n * sizeof(*p)    // Type of result: T*
```

Examples: (remember that **paddr(a)** is our address-printing macro)

```
paddr((char *)100 + 1);     // 101
paddr((short *)100 + 1);    // 102
paddr((int *)100 + 1);      // 104
paddr((long *)100 + 1);     // 108
```

Pointer arithmetic, continued

At hand:

If an integer is added to or subtracted from a pointer, the integer is "scaled" by the size of the pointed-to object.

What are the following values?

(char *)100 - 1

99

(short *)100 - 5

90

(int *)100 - 10

60

(double *)100 - 10

20

(long double _Complex *)100 - 1

68

Sidebar: *p++

The expression *p++ is an idiom that is important to understand.

First, note that *p++ means *(p++). p++ is evaluated first, producing p.

Consider this code that computes the length of a string:

```
char s[] = "test"; char *p = s; int len = 0;
while (*p++)
    len++;
```

Above, the expression *p++ is an L-value that specifies an element of s.

Before the first iteration of the loop, the L-values *p, *p++, and s[0] all specify the same memory object: the char that's the first element of s.

The compiler desires while (*R-value*) *statement*. Because *p++ is an L-value it fetches the value contained in the object, a char, and executes the body of the loop if that char value is non-zero.

Bottom line: The R-value of *p++ is the value contained in the memory object referenced by p. As a side-effect, p is "incremented".

Sidebar: *p++, continued

Here's another way to compute the sum of the elements in an array:

```
int sum(int a[], int nelems)  // ptrarith3.c
{
    int sum = 0;
    int *p = a;

    while (nelems--)
        sum += *p++;  // *(p++)

    return sum;
}
```

Usage:

```
int vals[] = { 10, 2, 4 };
iprint(sum(vals, 3));
iprint(sum(&vals[1], 2));
```

Less straightforward but entirely valid:

```
int rsum(int a[], int nelems)
{
    int sum = 0;
    int *p = &a[nelems]-1;

    while (nelems--)
        sum += *p--;   // *(p--)

    return sum;
}
```

Work through **rsum** with a diagram, too.

Exercise: Draw a memory diagram and work through a call to **sum**.

Sidebar: *p++, continued

Here is another formulation of the character-replicating **fill()** routine shown earlier:

```
void fill(char s[], char c, int n)
{
   char *p = s;

   while (n--)
      *p++ = c;  // *(p++)

   *p = 0;
}
```

p can be thought as a "flyweight" iterator: it maintains a position in a data structure.

Here is a previously-shown related example:

```
char s[] = "test";  char *p = s;  int len = 0;
while (*p++)
   len++;
```

*p++ iterates over the characters in s, producing a zero when the end is reached.

Subtraction of pointers, continued

A pointer can be subtracted from another pointer of the same type. The difference is scaled by the size of the pointed-to objects. Example:

```
int a[5];

int *p0 = &a[0];
int *p4 = &a[4];

paddr(p0);
paddr(p4);

printf("p4 - p0 = %td\n", p4 - p0);    // Use %td for pointer difference
printf("p0 - p4 = %td\n", p0 - p4);
```

Output:

```
p0 = 140734059040144
p4 = 140734059040160
p4 - p0 = 4
p0 - p4 = -4
```

Subtraction of pointers, continued

Speculate: What is the type of *pointer - pointer*?

The standard says,

> When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is **ptrdiff_t** defined in the **<stddef.h>** header.

Here's an example of using **ptrdiff_t**:

```
#include <stddef.h>
...
double x[50];
ptrdiff_t diff = &x[10] - x;
printf("difference: %td\n", diff);   // difference: 10
```

We'll learn all about **typedef**s later, but here is **ptrdiff_t** on lectura:

```
typedef long int ptrdiff_t;
```

Subtraction of pointers, continued

From the previous slide:

> "When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object..."

If the two pointers do not point to elements of the same array, the result is undefined.

```
int a[5];
int x, y, z;

&a[3] - &a[0]  // OK

&a[5] - &a[0]   // OK: a[5] is "one past"

&a[-1] - &a[0]  // Undefined: a[-1] is not in a or "one past"

&a[10] - &a[0]  // Undefined: a[10] is not in a or "one past"
```

Subtraction of pointers, continued

Here is a type algebra for pointer arithmetic:

T* + *integer* —> **T***

integer + **T*** —> **T***

T* - *integer* —> **T***

T* - **T*** —> signed integer, implementation specific (**ptrdiff_t**)

Adding two pointers is not permitted. Why not?

Consider a similiar problem:

What's July 2, 2015 minus June 1, 2014?

395 days

What's July 2, 2015 plus June 1, 2014?

Comparison of pointers

Pointers can be compared in various ways.

Two pointers of the same type can be tested for equality or inequality:

```
int *p1 = ..., *p2 = ...;

if (p1 == p2)
    ...

if (p1 != p2)
    ...
```

Comparison of pointers, continued

Comparing a pointer to an integer generates a warning unless the integer is a zero-valued constant.

```
if (p1 == 400)  // Warning: comparison between pointer and integer
  ...

if (p2 == 0)         // No warning
  ...

if (p3 == NULL)   // No warning
  ...
```

Pointers are often directly used to drive control structures:

```
if (!p1)
   ...

while ((p2 = findnext(val, values)))
   ...
```

Comparison of pointers, continued

The relational operators <, >, <=, and >= can be applied to pointers.

Here's yet another way to sum the elements in an array:

```
int sum(int a[], int nelems)
{
    int sum = 0;
    int *p = a;

    while (p < &a[nelems])
        sum += *p++;

    return sum;
}
```

Pointers being compared are subject to the same restriction as pointers that are being subtracted: they must reference elements in the same array, or one past the last element.

Pointers and arrays

Consider a loop that sets each element of an array to its index:

```
int a[10];

for (int i = 0; i < 10; i++)
    a[i] = i;
```

This loop produces the same result:

```
for (int i = 0; i < 10; i++)
    *(a + i) = i;
```

A VERY important rule:

If `a` is an array, the expressions

```
a[i]
```

and

```
*(a + i)
```

are completely equivalent.

This mechanical transformation can be applied to every array reference in any C program written, and it is guaranteed to work!

Pointers and arrays, continued

For reference, given `int a[10]`, these two loops are equivalent:

for (int i = 0; i < 10; i++)
    a[i] = i;
	for (int i = 0; i < 10; i++)
	    *(a + i) = i;

The examples below are equivalent to each other, and illustrate another rule. Speculate: What's the rule?

int *p = a;
for (int i = 0; i < 10; i++)
    *(p + i) = i;
	int *p = a;
	for (int i = 0; i < 10; i++)
	    p[i] = i;

The code above illustrates this rule:

If `p` is a pointer, the expressions

`*(p + i)`

and

`p[i]`

are equivalent.

<u>This mechanical transformation can be applied to any array reference in any C program, and it is guaranteed to work!</u>

Pointers and arrays, continued

A curious program:

```
int main()
{
   int i = 0;
   while (i < 7)
      putchar(i++["Hello!\n"]);
}
```

Output:

```
Hello!
```

For fun: Explain it!

For more code like this, see *Obfuscated C and Other Mysteries*, by Don Libes.

Revisiting: Arrays as function parameters

Recall that if an array is passed to a function, **sizeof** reports that the size is 8:

```
int f(int a[])
{
   psize(a);  // prints 8, regardless of actual argument passed
   ...
}
```

In fact, C considers an array parameter to simply be a pointer.

Here's how the compiler "sees" **f**:

```
int f(int *a)
{
   ...
}
```

Any one-dimensional array parameter in any function can be changed to a pointer.

Any pointer parameter in any function can be changed to a one-dimensional array.

Pointers and arrays, continued

Recall our **swap** function:

```
void swap(int *ap, int *bp)
{
   int t = *ap;
   *ap = *bp;
   *bp = t;
}
```

Problem: Sadly, your 8/* key stopped working just before you started to type in the above. Rewrite **swap** without using an asterisk.

Solution:

```
void swap(int a[], int b[])
{
   int t = a[0];
   a[0] = b[0];
   b[0] = t;
}
```

String manipulation using pointers

String manipulation using pointers

String manipulation routines are most commonly written using pointers rather than array indexing.

Comparing array-based and pointer-based versions of the same routine provides insight into the shift in thinking that should take place when you grasp the full power of pointers.

Here are array-based and pointer-based versions of **pstring**:

```
void pstring(char s[])
{
    for (int i = 0; s[i]; i++)
        putchar(s[i]);
}
```

```
void pstring(char *p)
{
    while (*p)
        putchar(*p++);
}
```

p [ ]

t	e	s	t	i	n	g	\0

100 ... 106

Remember that a pointer can be thought of as a flyweight iterator that specifies a position in a sequence of values.

String manipulation using pointers, continued

From the previous slide:

void pstring(char s[])
{
   for (int i = 0; s[i]; i++)
      putchar(s[i]);
}
	void pstring(char *p)
	{
	   while (*p)
	      putchar(*p++);
	}

It's important to "think pointers". If you're still thinking in terms of arrays, but try to use pointers, you might end up with a **pstring** like this:

```
void pstring(char *p)
{
   for (int i = 0; *(p + i); i++)
      putchar(*(p + i));
}
```

String manipulation using pointers, continued

Here's the **pos** routine written in both styles. Recall that **pos** returns the 0-based position of the first occurrence of a character in a string.

```
int pos(char s[], char c)
{
    for (int pos = 0; s[pos]; pos++)
        if (s[pos] == c)
            return pos;

    return -1;
}
```

```
int pos(char *p0, char c)
{
    for (char *p = p0; *p; p++)
        if (*p == c)
            return p - p0;

    return -1;
}
```

Note that when **p** is pointing at an occurrence of the character of interest, pointer subtraction is used to calculate an integral position to return.

Because parameters that are pointers are completely equivalent to parameters that are arrays, the array-based version could be declared as **int pos(char *s, char c)** and the pointer-based version could be declared as **int pos(char p0[], char c)**.

How could we turn it into **char *pos(char *p0, char c)**? Would that be better?

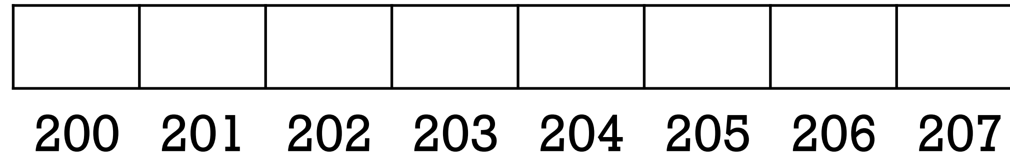
String manipulation using pointers, continued

Here are two ways to copy a string:

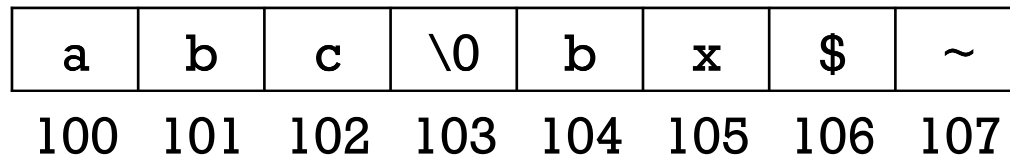
```
void copy(char to[], char from[])
{
    int i = 0;
    while ((to[i] = from[i]))
        i++;
}
```

```
void copy(char *to, char *from)
{
    while ((*to++ = *from++))
        ;
}
```

to



from



String manipulation using pointers, continued

Here's string concatenation with arrays and pointers:

```
void concat(char to[], char from[])
{
  int topos = 0;

  while (to[topos])
    topos++;

  int frompos = 0;
  while (to[topos++] = from[frompos++])
    ;
}
```

```
void concat(char *to, char *from)
{
  while (*to)
    to++;

  while (*to++ = *from++)
    ;
}
```

to

p	o	i	n	t	\0	\0	.	8	2	=

from

i	n	g	\0	u	p	\0	}

String manipulation using pointers, continued

Here's another array-based version of **concat**:

```
void concat(char to[], char from[])
{
 int i = 0, j = 0;
 while (to[i])
   i++;

while (to[j++] = from[i++])
   ;
}
```

Is it correct?

No. **i** and **j** are transposed in the second **while**!

Note that the pointer-based version eliminates the possibility of mismatching arrays and indices. A pointer is inseparable from the data it references!

Another lesson from above: Use better variable names!

A little more on string literals

Consider this declaration:

```
char *s = "abc";
```

Remember: a string literal is syntactic sugar for an array initialization. A simple view of the declaration above is this,

```
char _tmp918[] = { 'a', 'b', 'c', 0 };
char *s = _tmp918;
```

but the following is more accurate:

```
const char _tmp918[] = { 'a', 'b', 'c', 0 };
char *s = _tmp918;
```

The **const** *qualifier* specifies that the values in the array should not be changed. The C compiler is then free to direct that the array be placed in a region of memory that the operating system treats as read-only. (We'll learn more about **const** later.)

The following code produces a segmentation fault:

```
char *s = "abc";
*s = 'X';
```

Because the literal string is placed in read-only memory, the assignment, which would change the 'a' to an 'X', generates a fault.

A little more on string literals, continued

Calling a function and specifying a string literal for a string argument that is modified by the function leads to a fault.

Consider a **copy** call with reversed operands:

```
char buf[20] = "x";
copy("testing", buf);
```

The above code is roughly equivalent to this:

```
char buf[20] = {'x', 0 }
const char _tmp623[] = {'t', 'e', 's', 't', 'i', 'n', 'g', 0 };  // Note the const
copy(_tmp623, buf);
```

Essentially, the first action performed by **copy** is **_tmp623[0] = buf[0]** and because **tmp** is in a block of memory marked as read-only by the operating system, a fault is generated.

Exercise: The above code is in **lit2.c**. Compile with **-S** to generate assembly code, and then look for **testing** in **lit2.s**. You'll that it's preceded by "**.section rodata**", which directs that data, the string "**testing**", to be in read-only memory.

A little more on string literals, continued

Executive summary:

As a rule, string literals cause the creation of immutable arrays of **char** values. The exception is when a string literal is used to initialize a **char** array.

Example:

```
char text1[] = "some text";  // text1[i] can be changed, but
                             // text1 cannot be changed.

text1[0] = 'x';              // OK
text1++;                     // NOT ALLOWED


char *text2 = "more text";   // text2[i] can't be changed, but
                             // text2 can be changed.

text[0] = 'x';               // NOT ALLOWED
text2++;                     // OK—*text2 is 'o'
```

A common error

Here is a very lucky program:

```
int main()
{
   char *s;

   strcpy(s, "testing");
   strcat(s, " this");

   printf("s = '%s'\n", s);
}
```

Output:

```
s = 'testing this'
```

Confession:
This example used to "work" but no longer does.

Challenge:
Leave **char *s;** unchanged and add a declaration before it that does make this example work. (See slides 242-243 for a potentially useful technique.)

What's lucky about it?

s is an uninitialized pointer! **s** apparently holds junk that's a valid address.

gcc does warn that **s** is uninitialized but in a more complex case it might not be detected.

Another common error

Consider the following code.

```
char *replicate(char c, int n)
{
    char result[n+1];
    char *p = result;

    while (n--)
        *p++ = c;
    *p = 0;
    return result;
}

int main()
{
    char *xs = replicate('x', 10);
    int len = strlen(xs);
    printf("xs = '%s' (%d)\n", xs, len);
}
```

Some runs:

```
% a.out
xs = '`m#{#  ' (10)

% a.out
xs = '##&9#######0)

% a.out
xs = '`###  ' (10)

% a.out
xs = '' (10)
```

What's wrong?

Another common error, continued

We're getting junk from this:

```
char *replicate(char c, int n)
{
    char result[n+1];
    char *p = result;

    while (n--)
        *p++ = c;
    *p = 0;
    return result;
}

int main() // lucky4.c
{
    char *xs = replicate('x', 10);
    int len = strlen(xs);
    printf("xs = '%s' (%d)\n", xs, len);
}
```

We do get a **gcc** warning: **function returns address of local variable**

The array **result** is a local variable in **replicate**.

Because **result** is a local variable, it has automatic storage duration.

Space for **result** is reserved on the stack for the lifetime of **replicate**, but after **replicate** returns, that space can be used by another function!

It seems that **result** is getting clobbered sometime after **strlen** computes the length but before **printf** outputs the first character of **result**, but how could we investigate further?

Use **gdb**!

Returning the address of a local is always a mistake

In general, **it's always a mistake to return the address of a local** because the contents of that local are undefined after a function returns.

```
int *f(...)
{
    int a[10], x;
    ...

    if (...)
        return a;
    else if (...)
        return &a[n];
    else
        return &x;
}
```

The memory where **a** and **x** reside is "up for grabs" as soon as **f** returns!

In C we must always be always be cognizant of object lifetimes, lest we use a corpse!

Is it ok?

Is it ok to directly return an element from a local array?

```
int f()
{
   int a[10];
     ...
   return a[n];    // Ok?
}
```

Yes, **a[n]** is simply a value. There's no concept of lifetimes with values.

Is it ok?

It is ok to pass the address of a local array, or an element in a local array to a function?

```
int f2(...)
{
   int a[10], x;
   ...
   g(a);          // Ok?
   g(&a[2]);      // Ok?
   g(&x);         // Ok?
}
```

The above calls are ok because `a` and `x` live until `f2` returns, and `f2` can't return until all of the calls to `g` have returned.

Is it ok to return a pointer difference of two locals?

```
...in f2 above...
return &a[i] - &a[j];
```

Yes. We're relying on the distance between the values, not the values themselves.

Arrays of pointers

Understanding a declaration

The first step in understanding arrays of pointers is to understand the declaration of an array of pointers.

Consider this declaration:

```
char *a[3];
```

By virtue of "declaration mimics use" we know this:

```
*a[n]    is    char
```

Let's remove the lowest precedence operation from the left column and prepend an English equivalent in the right-hand column:

a[n]   is   <u>pointer to</u> char

Let's repeat the process, reaching only an identifier:

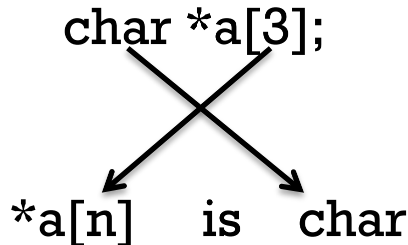
a   is   <u>array of</u> pointer to char

Thus, `char *a[3]` declares `a` to be an array of three pointers that reference `char`s.

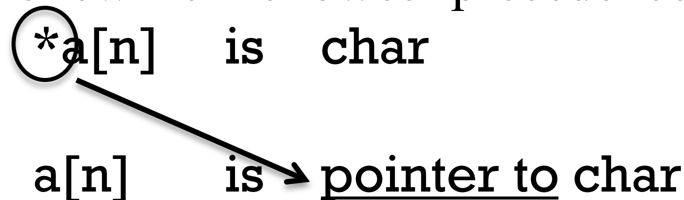
Understanding a declaration, continued

Let's review that process.

Here's the declaration. Swap the two parts and put "is" in the middle.

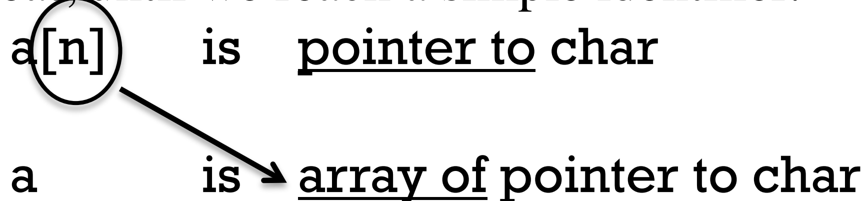


Let's rewrite the lowest-precedence operator in English



Note that the intermediate form is correct, too: if we see **a[n]**, we know it is a **pointer to char**.

Repeat, until we reach a simple identifier.



Some systems have **cdecl(1)**, but **cdecl** is web-enabled at **cdecl.org**.

Understanding a declaration, continued

In summary, the declaration

```
char *a[3];
```

shows us three ways to use a:

```
*a[n]   is   char
 a[n]   is   pointer to char
 a      is   array of pointer to char
```

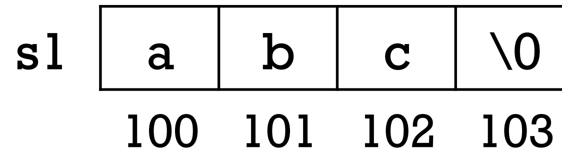
The following expressions are valid:

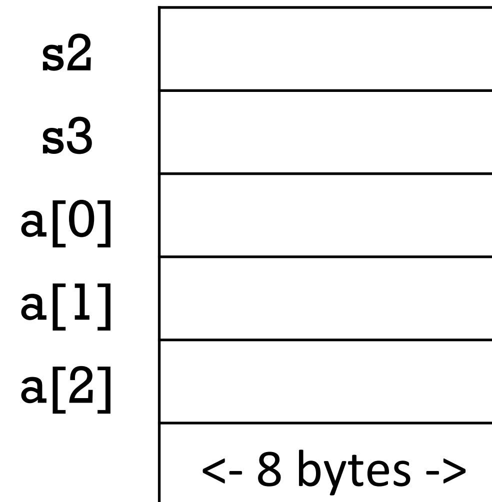
```
a[0] = "testing";

int len = strlen(a[0]);

*a[0] = 'x';

putchar(*a[0]);
```

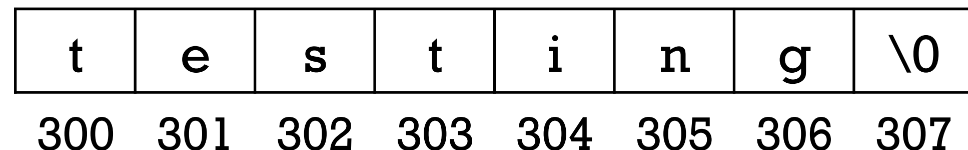
Example

Code:

```
int main()
{
    char s1[] = "abc";
    char *s2 = "testing";
    char *s3 = s2 + 4;

    char *a[3];
    a[0] = s1; // &s1[0]
    a[1] = s2;
    a[2] = ++s3;

    for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++)
        printf("%s (%lu chars)\n", a[i], strlen(a[i]));
}
```







Output:

```
abc (3 chars)
testing (7 chars)
ng (2 chars)
```

Each of the three values in **a** is the address of a zero-terminated sequence of char values.

Initializing an array of pointers

An array of pointers can be initialized with a sequence of values:

```
char *a[] = { "XX", "XIX", "XVIII" };

for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++) // ap1a.c
    printf("%s\n", a[i]);
```

Output:

```
XX
XIX
XVIII
```

Memory might look like this:

a[0]	300
a[1]	303
a[2]	307
	<- 8 bytes ->

X	X	\0	X	I	X	\0	X	V	I	I	I	\0	...
300	301	302	303	304	305	306	307	308	309	310	311	312	

Initializing an array of pointers, continued

It's common to see a null pointer mark the end of an array of pointers. I've grown to prefer using 0 rather than **NULL**.

```
char *s = "abcd";
char *a[] = { "XX", "XIX", "XVIII" };
char *b[] = { s, s+1, &s[2], a[1], a[2]+2, 0 };

for (int i = 0; b[i]; i++)
    printf("%s\n", b[i]);
```

Output:

s	a	b	c	d	\0
	100	101	102	103	104

b[0]	
b[1]	
b[2]	
b[3]	
b[4]	
b[5]	
b[6]	

a[0]	300
a[1]	303
a[2]	307
	<- 8 bytes ->

X	X	\0	X	I	X	\0	X	V	I	I	I	\0	...
300	301	302	303	304	305	306	307	308	309	310	311	312	

Exercise: Do this with gdb, too!

Sidebar: Lots of ways to handle array lengths

We've now got four distinct ways of dealing with array length!

1. If an array is a local or a global, we can use **sizeof(a)/sizeof(a[0])**, or the slightly shorter **sizeof(a)/sizeof(*a)**.

2. If an array is passed to a function, all that's passed is the address of an element. Unless there's a sentinel value, which is uncommon with numeric quantities, an element count must be passed, too.

3. If we've got a **char** array that represents a C string, we can assume zero-termination. **This is the only case where strlen() works!**

4. With arrays of pointers, a null pointer (0 or **NULL**) is a perfect sentinel value because C guarantees that no object will ever be assigned that address (which is usually 0.)

Command line arguments

Now (and only now!) can we fully understand a C program that prints the command line arguments:

```
% cat args.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++)
        printf("|%s|\n", argv[i]);
}

% gcc args.c && a.out just ' "testing" '   "t h i s"
|a.out|
|just|
| "testing" |
|t h i s|
```

Another way: (late addition)

```
int main(int argc, char *argv[])
{
    for (char **ap = argv; *ap; ap++)
        printf("|%s|\n", *ap);
}
```

Note to self: above uses char **--too soon!

Note that the first element (`argv[0]`) is the program name, not the first argument.

Command line arguments, continued

I lied a little. I actually see the following. Why?

```
% a.out just '"testing"' "t h i s"
|./a.out|
|just|
|"testing"|
|t h i s|
```

We'll see in argv[0] whatever path a program is run with:

```
% gcc -o args args.c
% $fall15/c/args hello!
|/cs/www/classes/cs352/fall15/c/args|
|hello!|
```

What's something interesting we could do by looking at the program name?

We could vary program behavior based on program name.

We can use a symlink to run a program using a different name:

```
% ln -s args hello
% hello
|hello|
```

Pointers to pointers

Pointers to pointers

Given `char *a[3]`, let's consider the expression `a`:

- Because `a` is an array, `a` is equivalent to `&a[0]`.
- The type of `a[0]` is pointer to `char`.
- The type of `&a[0]` is pointer to pointer to `char`.

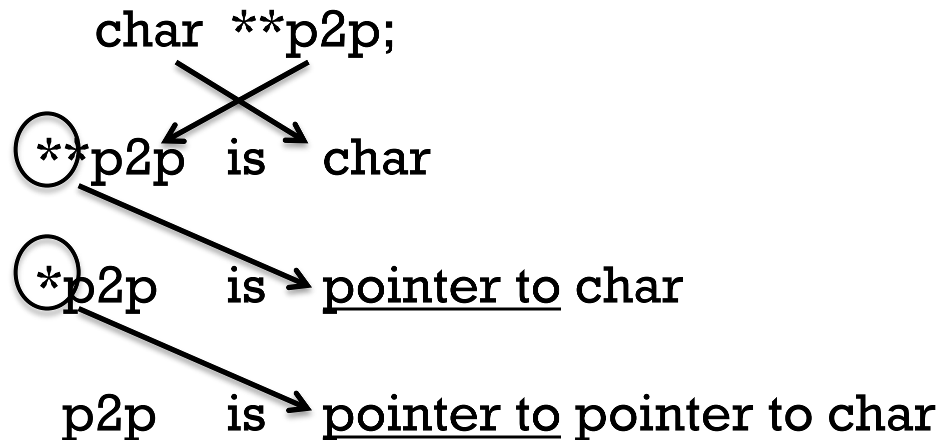
The expression `a` is a pointer to a pointer. It is the address of an address that holds a `char`.

Pointers to pointers, continued

Here is a declaration for a pointer to a pointer:

```
char **p2p;
```

To understand the type of `p2p`, we can swap the two parts, and then repeatedly remove the lowest precedence operation from the left column, and prepend an English equivalent in the right-hand column, until reaching a simple identifier.



Pointers to pointers, continued

Let's use a pointer to a pointer to step through an array of character pointers.

```
int main()
{
    char *a[] = { "one", "two", "three"};

    char **p2p;
    p2p = a;

    char *p1 = *p2p;
    puts(p1);
    p2p++;
    puts(*p2p);
    puts(*++p2p);
    printf("%td\n", p2p - a);
}
```

For reference:

```
char **p2p;

**p2p is  char
*p2p  is  pointer to char
 p2p  is  pointer to pointer to char
```

		address
a[0]	300	100
a[1]	304	108
a[2]	308	116
p2p		124
p1		132
	<- 8 bytes ->	

o	n	e	\0	t	w	o	\0	t	h	r	e	e	\0
300	301	302	303	304	305	306	307	308	309	310	311	312	313

Pointers to pointers, continued

Imagine a function that returns a pointer to the pointer to the longest string in a zero-terminated array of **char** pointers:

```
char *a[] = { "Give", "me", "more", "pointers", "please!", 0 };
char **lng = longest(a);
printf("The longest word is '%s'\n", *lng);
```

Implementation:

```
char **longest(char **strings) // ptr2ptr2.c
{
    char **longest = 0, **p2p;

    for (p2p = strings;  *p2p;  p2p++)
        if (longest == 0)
            longest = p2p;
        else if (strlen(*p2p) > strlen(*longest))
            longest = p2p;

    return longest;
}
```

What's a simple efficiency improvement?
Store **strlen(*longest)**, too.

Pointers to pointers, continued

Just as **f(char *p)** and **f(char a[])** are equivalent, so are **f(char **p)** and **f(char *a[])**. Here's another way to write **longest**, albeit less idiomatic:

```
char **longest(char *strings[])
{
    char **longest = 0;

    for (int i = 0;  strings[i];  i++)
        if (longest == 0)
            longest = &strings[i];
        else if (strlen(strings[i]) > strlen(*longest))
            longest = &strings[i];

    return longest;
}
```

Both definitions of **longest** can use

```
char **longest(char *strings[]) ...
```

or

```
char **longest(char **strings) ...
```

Example: a simple `tac`

Potential representation

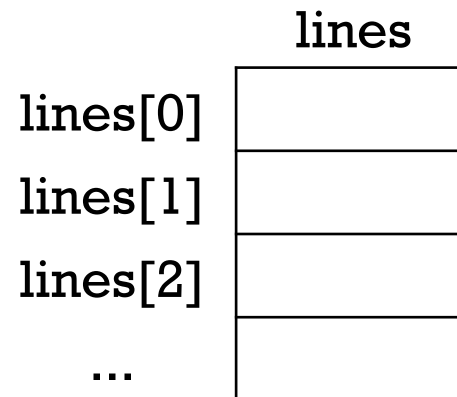
Recall that **tac(1)** reads lines from standard input and prints them in reverse order, first line last, last line first.

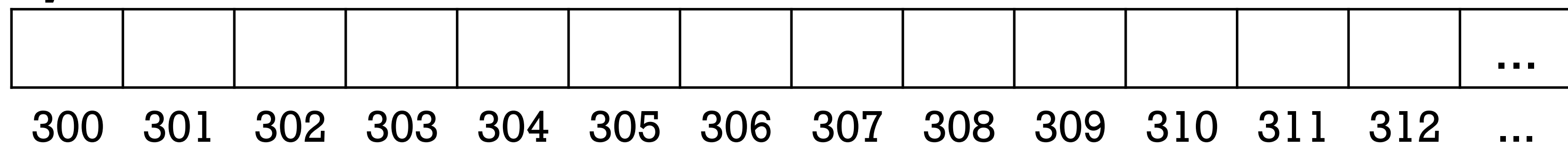
Let's implement a simplified version of **tac**. It has two limitations: its input must be less than **MAX_BYTES** in length and can have no more than **MAX_LINES** lines.

I have a representation in mind that uses two arrays:

```
char *lines[MAX_LINES];
char bytes[MAX_BYTES];
```

What have I got in mind?





Potential representation, continued

At hand:

```
char *lines[MAX_LINES];
char bytes[MAX_BYTES];
```

My idea: Lets store the content of input lines consecutively in **bytes**, and store in **lines** a series of pointers to the start of each line contained in **bytes**.

Here's a simple input:

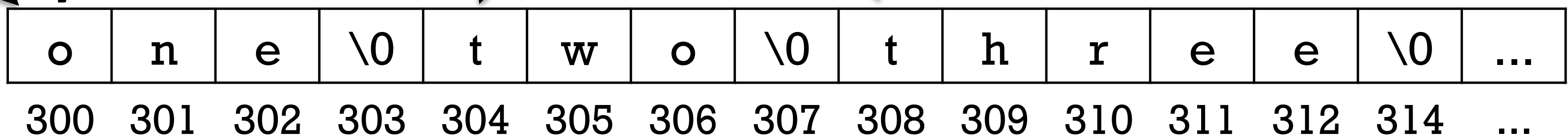
```
% cat tac.1
one
two
three
```

Here's snapshot of memory after reading those lines:

lines

	lines
lines[0]	300
lines[1]	304
lines[2]	308
...	

bytes

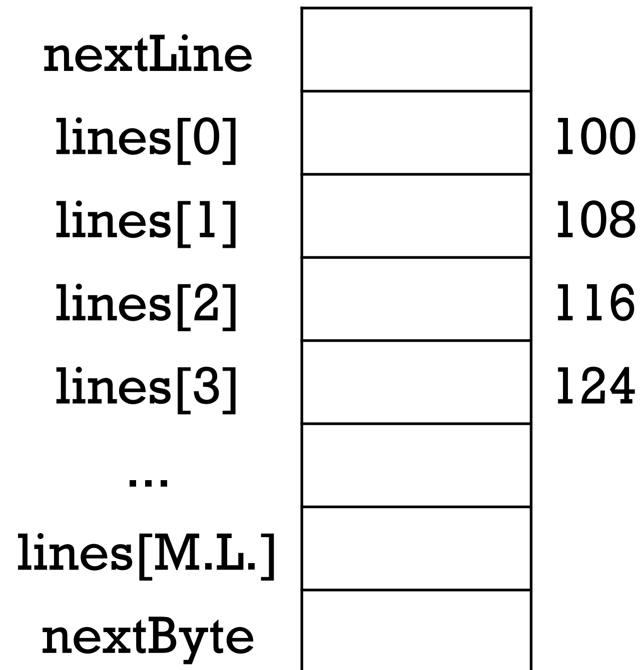


Implementation

```
#define MAX_LINES 10000
#define MAX_BYTES 1000000
int main()  // tac.c
{
   char *lines[MAX_LINES],
   char **nextLine = lines;
   char bytes[MAX_BYTES];
   char *nextByte = bytes;

   while (gets(nextByte)) {
      *nextLine++ = nextByte;
      nextByte += strlen(nextByte) + 1;
      }

   while (nextLine > lines)
      puts(*--nextLine);
}
```

Input:
one
two
three

Output:



bytes

Refinement: assertions

Let's warn the user if our assumptions are violated.

```
#include <assert.h> // get the assert macro
...
int main()
{
    char *lines[MAX_LINES];  char **nextLine = lines;
    char bytes[MAX_BYTES];  char *nextByte = bytes;

    while (gets(nextByte)) {
        assert(nextLine < lines + MAX_LINES);
        *nextLine++ = nextByte;
        nextByte += strlen(nextByte) + 1;
        assert(nextByte < bytes + MAX_BYTES);
        }
    ...
}
```

Interaction:

```
% seq 10001 | a.out
a.out: tac.c:16: main: Assertion 'nextLine < lines + 10000' failed.
Aborted (core dumped)

% yes $(seq 37) | head -c1000000 | a.out
a.out: tac.c:19: main: Assertion 'nextByte <= bytes + 1000000' failed.
Aborted (core dumped)
```

Pointers to pointers to pointers to ...

It's very common to see both pointers, and pointers to pointers in C code but pointers to pointers to pointers (and beyond) are relatively uncommon.

Here is a trivial example that uses a **char** ***:

```
int main()
{
   char *a[] = { "Give", "me", "more", "pointers", "please!", 0 };
   char *b[] = { "More", "Goooooooogle", "results", 0 };

   char **ap[] = { a, b }; // array of pointer to pointer to char

   char ***p = ap;         // pointer to pointer to pointer to char

   printf("%c\n", ***p);       // G
   printf("%s\n", **p);        // Give
   printf("%s\n", *p[1]);      // More  (Note that *p[1] is *(p[1]).)
   printf("%c\n", **p[1]);     // M
}
```

void *

Imagine a generalized version of **swap** that takes two addresses and swaps N bytes at those addresses:

```
int i = 10, j = 20;
swap(&i, &j, sizeof(int));

double x = 12.34, y = 56.78;
swap(&x, &y, sizeof(double));
```

What should be the type of **swap**'s first two arguments?

swap(int *a, int *b, int nbytes)?

swap(double *a, double *b, int nbytes)?

swap takes two pointers to "something". We can use **void *** to express that:

```
swap(void *a, void *b, int nbytes)
```

void *, continued

void * gives us a way to say "pointer to something".

Given **void *vp**, any pointer can be assigned to **vp**. No cast is needed.

The following code generates no warnings:

```
int main(int argc, char *argv[])
{
    double a[3];
    void *vp;

    vp = &argc;       // pointer to int
    vp = a;           // pointer to double
    vp = "abc";       // pointer to char
    vp = argv;        // pointer to pointer to char
    vp = argv[1];     // pointer to char
    void *ptrs[] = { &argc, a, "abc", argv, argv[1] };
}
```

At present, it is common for all pointer types to be the same size, but if pointers are different sizes, **void *** must be big enough to represent the largest.

Fine point: a **void *** like **vp** above may actually hold both a pointer and type information that indicates what kind of pointer is being held.

void *, continued

The previous slide showed that a pointer of any type can be assigned to a **void *** without needing a cast.

Similarly, a **void *** can be assigned to a pointer of any type without needing a cast.

```
void *vp;
int *ip;
int i;

vp = &i;
ip = vp;
```

What's something that we can't do with **void *vp**?

We can't indirect through it. ***vp** isn't valid! Why not?

How big is the memory object specified by the L-value ***vp**?

Is **vp++** valid?

Yes, but that surprises me!

Memory allocation

What's the problem we're about to solve?

C's fixed-size arrays are at odds with a common need of programs: holding an arbitrary number of things.

Our simple `tac` program has limits specified with `#define`s, and those could be raised, but should `tac` really be subject to any limits?

`tac` should be able to reverse any input that will fit in memory.

What's the maximum resolution that an image editor should be able to handle?

It should be able to handle any image that will fit in memory.

What limits should a browser place on the number of elements in an HTML document?

If an HTML document and its graphical representation fit in memory, a browser should be able to display it.

A general principle of industrial-strength software design:

A program should be able to utilize all available memory to meet the needs of a user.

The problem, continued

How do Java programs accommodate indefinitely large inputs?

- Instances of classes like **`ArrayList`** expand as needed.
- Methods like **`BufferedReader.readLine()`** can read arbitrarily long lines of input.
- Data structures like linked lists and trees can extend arbitrarily; we just create new nodes with **`new`** and add them to the structure as needed.
- Simple arrays are not expandable, but a new, larger array can be allocated, and elements can then be copied in from an old array.

These very same techniques work in C but there's an additional burden:

When we're done using a block of memory, we need to explicitly release it.

The third type of storage duration

We've studied two types of storage duration in C:

Automatic

Local variables have automatic storage duration. When a block of code is entered, space is reserved for locals declared in the block. When the block is exited, that space is reclaimed.

Static

Global variables and static locals have static storage duration. Space is reserved for globals and static locals before execution begins, and not released until the program terminates.

The third type of storage duration is "allocated".

- The lifetime of allocated storage is controlled by the program.
- When memory is needed, it is allocated by calling a library routine.
- When allocated memory is no longer needed, it is released by calling a library routine.

The C11 standard provides four functions for memory allocation: **malloc**, **calloc**, **realloc**, and **aligned_alloc**. We'll study the first three, and some additional functions that are outside the standard, such as **strdup** and **asprintf**.

malloc

The most commonly used allocation function is **malloc**. The standard says this:

7.22.3.4 The malloc function

Synopsis

```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

The **malloc** function allocates space for an object whose size is specified by **size** and whose value is indeterminate. [*Recall that* ***size_t*** *is a* ***typedef*** *for* ***unsigned long****. We'll think of it as an* **int** *for now.*]

Returns

The **malloc** function returns either a null pointer or a pointer to the allocated space.

Note that **malloc**'s return type is **void ***, expressing that the memory is thought of as not holding an object of any particular type. Note the the object (singular) may be an array.

What does "...whose value is indeterminate." mean?

malloc, continued

For reference:

```
void *malloc(int nbytes);
```

Let's request a block of allocated memory that is large enough for 25 **int** values:

```
int *p = malloc(25 * sizeof(int));  // Note: no (int *) cast required
printf("p = %lu (%p)\n", p, p);
                        // Output: p = 31002640 (0x1d91010)
```

Let's fill the newly allocated memory with some values:

```
int *p2 = p, i = 1;
while (p2 < p + 25)
   *p2++ = i++ * 2;
```

Let's see what **gdb** shows, including a little extra on each side:

```
(gdb) p p[-2]@29
$1 = {113, 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38,
40, 42, 44, 46, 48, 50, 0, 135057}
(gdb) p p2-p
$2 = 25
(gdb) p *p2
$3 = 0
```

malloc, continued

It's common to have a function return a pointer to a block of allocated memory that holds the result of a computation. Example:

```
char *replicate(int n, char c)
{
   char *buf = malloc((n+1)*sizeof(char));
   char *p = buf;

   while (n--)
      *p++ = c;
   *p = 0;

   return buf;
}
```

Usage:

```
char *As = replicate(5, 'A');
char *dots = replicate(3, '.');

printf("%s%s%s\n", As, dots, As);
 // Output: AAAAA...AAAAA
```

About **malloc((n+1)*sizeof(char))**...

Why **(n+1)**?

Allows space for terminating zero.

Is **sizeof(char)** really needed?

No, but it makes the intent clear.

Are both **buf** and **p** really needed?

malloc, continued

For reference:

```
int *p = malloc(25 * sizeof(int))
```

The Java analog:

```
int p[] = new int[25];
```

What's a key difference between the two?

Java can detect a mismatch like this:

```
int a[] = new char[n];
```

C won't detect this:

```
int *a = malloc(n*sizeof(char));
```

Another difference: we don't have to put **int**s in the memory allocated by **malloc(25*sizeof(int))**. We could use it to store 100 **char**s instead; it's just a chunk of memory!

malloc, continued

For reference:

```
int *p = malloc(25 * sizeof(int))
```

Just as we can go outside the bounds of an array, we can go outside the bounds of an allocated block. The following code will likely execute without an immediate error:

```
int x = p[25];
p[-1] = 100;
p[25] = 200;
```

There are numerous designs for memory allocators. In some allocators, data structures used by the allocator immediate precede or follow the region of memory given to the user. An overrun or underrun might corrupt the allocator's data structures.

As with any other stray read or write to memory, out of bounds references with blocks of allocated memory might not manifest themselves for a long time, if ever.

malloc, continued

Are there any errors in the following code?

```
int *p = malloc(10);  // Get space for ten ints
for (int i = 0; i < strlen((char*)p); i++)
    p[i] = i;
```

Despite the comment, we are actually allocating space for ten **char**s, not ten **int**s!

What should it be?

```
int *p = malloc(10*sizeof(int));
```

What will **strlen((char*)p)** produce?

It will count bytes until it finds a zero in the indeterminate values in memory starting at the address in **p**!

Should we use **sizeof(p)** instead?

No, that will produce 8, the size of an **int ***!

Fact: There's no reliable mechanism to query the size of a block of memory allocated with **malloc**.

The **free** function

Memory that we get with **malloc** can be returned with **free**.

Here's a trivial "round-trip"—we get a block of memory for ten **int**s and then immediately give it back:

```
int *p = malloc(10*sizeof(int));
free(p);
```

Note that while we specify an amount of memory to allocate, we don't specify an amount to free. Instead, the allocator keeps track of the size of all allocated blocks and given an address, knows the size of a block at an address.

Why is it good to not need to specify a size for **free()**?

Imagine this code:

```
int *p = malloc(n*sizeof(int));
...populate p with values...
f(p); // f will call free(p) when done
```

If **free()** needed a size, we'd need to pass it along, too: **f(p, n*sizeof(int))**.

free, continued

This C loop will run forever:

```
for (;;) {
    int *p = malloc(1000 * sizeof(int));
    free(p);
    }
```

Estimate: How many iterations per second of the above loop on lectura?

About 20 million

How much slower would it be with **malloc(100000 * sizeof(int))**?

Find out, with **malloc3.c**!

Does every call to **malloc** return the same address?

It appears so.

What happens if we remove **free(p)**?

It eventually consumes all memory available to the process.

malloc returns 0 if the requested memory can't be allocated, but there's more to the story.

free, continued

For reference: (20 million iterations/second)

```
for (;;) {
    int *p = malloc(1000 * sizeof(int));
    free(p);
    }
```

A Java analog:

```
for (;;) {
        int p[] = new int[1000];
        }
```

Estimate: How many iterations per second?

About 1.4 million

Why doesn't the Java version run out of memory?

Java provides automatic memory management, also known as *garbage collection*: when an allocated object can no longer be accessed its memory is subject to reclamation.

Java vs. C

Memory management in C and Java are polar opposites.

Java:

A program allocates memory as needed. Java's garbage collector is responsible for recycling objects that can no longer be reached.

C:

A program allocates memory as needed but is also responsible for returning memory when no longer needed.

On almost all modern operating systems, the operating system reclaims all memory in use by a program when the program terminates.

Simple enough, right?

Memory management in C seems simple at first glance:

Free allocated memory when you're done with it.

However, there are several facets to consider:

- *If memory is allocated it must be freed.*
- *Don't use memory after it is freed.*
  Corollary: Don't free memory that will be used later.
- *Don't free a block of memory more than once.*
- *Don't free memory that wasn't allocated.*

And of course:

Don't use memory outside the bounds of an allocated block.

The fact of it:

It is very difficult to avoid memory management errors in large C programs.

Memory leaks

A very common type of memory management error is a *memory leak*. A memory leak occurs when memory is no longer needed but is not freed.

Here is an obvious memory leak:

```
for (;;) {
    int *p = malloc(100 * sizeof(int));
    }
```

Is there a leak in the following code?

```
for (int i = 1; i <= 20; i++)
    printf("%s\n", replicate(i, 'A'));
```

```
char *replicate(int n, char c)
{
    char *buf =
        malloc((n+1)*sizeof(char));
    char *p = buf;
    while (n--)
        *p++ = c;
    *p = 0;
    return buf;
}
```

Yes, the memory allocated by **replicate** is never freed. Here's a fix:

```
for (int i = 1; i <= 20; i++) {
    char *s = replicate(i, 'A');
    printf("%s\n", s);
    free(s);
    }
```

But now the user of **replicate** is burdened. Could **replicate** do the **free** itself?

Memory leaks, continued

Here's a coding practice to avoid leaks that's practical in some cases:

When you write a call to **malloc()***, immediately write the corresponding call to* **free()**.

What's a case when that practice is not applicable?

When a function returns an object in allocated memory, like with **replicate**.

Solution: write the **free** as soon as you write **replicate(...)**.

If a word processor leaks 1000 bytes every time you save the document, how likely are you to notice that?

As a rule, the bigger and/or more frequent a leak, the easier it is to find.

A memory leak should never directly cause a program malfunction but a malfunction will often result if memory is exhausted.

Can there be memory leaks in a Java program?

"Used after freed"

If a function returns a pointer to allocated memory, the usual practice is to save the pointer, use the memory, and then free it. Recall:

```
for (int i = 1; i <= 20; i++) {
    char *p = replicate(i, 'A');
    printf("%s\n", p);
    free(p);
    }
```

A common error is to free the memory, then use it. Example:

```
for (int i = 1; i <= 20; i++) {
    char *p = replicate(i, 'A');
    free(p);
    printf("%s\n", p);
    }
```

This type of error is commonly called "used after freed".

What are likely behaviors for the above code? Is a fault likely?

"Double free"

Another common memory management error is to free a block of memory twice, typically due to confusion over which function is responsible for deallocation.

```
void f()
{
   char *p = replicate(10, 'x');
   g(p);
   free(p);
}

void g(char *p)
{
   printf("f(%s)\n", p);
   free(p);
}
```

This sort of error is often called a "double free" or "multiple free".

A different manifestation of confusion of responsibility is a leak—**f** and **g** might both assume that the other routine is to free the memory.

"Double free", continued

For reference:

```
void f()
{
   char *p = replicate(10, 'x');
   g(p);
   free(p);
}

void g(char *p)
{
   printf("f(%s)\n", p);
   free(p);
}
```

2005 behavior: Ran ok with gcc on lectura. Faulted on Windows XP.

Current behavior with gcc on lectura:

```
g(xxxxxxxxxx)
*** glibc detected *** ./a.out: double free or corruption (fasttop):
0x0000000000a32010 ***
======= Backtrace: =========
```

Freeing non-allocated memory

Another type of error is to free memory that wasn't allocated. Here's a goofy case that once ran without error with gcc on lectura:

```
char x[10];
free(x);
```

A more subtle error is to adjust a pointer that was initially provided by malloc:

```
int *p = malloc(...);
...
if (...) {
    ...
    *p++ = ...
    ...
    }

free(p);
```

It's only valid to call **free** with an exact address produced by **malloc**. In the above case, if **malloc** returns 1000 and the **if** block is executed, **free** is called with 1004, not 1000. I sometimes call this "Freeing an advanced pointer."

malloc implementations vary

The C11 standard simply says that behavior is undefined for memory management errors. The response of allocators to errors varies by implementation.

A simple example of freeing non-allocated memory:

```
int *p = malloc(10*sizeof(int));
free(p+1);
```

Current behavior on lectura:

```
% ./a.out
*** glibc detected *** ./a.out: free(): invalid pointer: 0x02541014 ***
======= Backtrace: =========
...
```

Current behavior on OS X with clang 6.0:

```
% ./a.out
a.out(65266,0x7fff74eec300) malloc: *** error for object
0x7f9600c04d34: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

Adjusting **malloc**'s behavior

From the **malloc** man page:

When [the *environment variable*] **MALLOC_CHECK_** is set, a special (less efficient) implementation is used which is designed to be tolerant against simple errors...

If **MALLOC_CHECK_** is set to 0, any detected heap corruption is silently ignored; if set to 1, a diagnostic message is printed on stderr; if set to 2, **abort(3)** is called immediately; if set to 3, a diagnostic message is printed ... and the program is aborted.

One or more environment variables can be set for the duration of a command by specifying *VAR=VALUE* at the start of a command line.

```
% MALLOC_CHECK_=0  a.out
% MALLOC_CHECK_=1  a.out
*** glibc detected *** ./a.out: free(): invalid pointer: 0x01b81014 ***
% MALLOC_CHECK_=2  a.out
Aborted (core dumped)
% MALLOC_CHECK_=3  a.out
*** glibc detected *** ./a.out: free(): invalid pointer: 0x0000000000c30014
***
======= Backtrace: ========= ...
```

Use **echo $MALLOC_CHECK_** to see if you've got it set. In your **~/.bashrc** you can use **export MALLOC_CHECK_=1** to set it "permanently". (See **man getenv**, too.)

World's simplest (and fastest) **malloc** and **free**!

Allocators are typically very complex, but it's not hard to write a simple allocator. Here are a simple **malloc** and **free** that are also, I claim, the world's fastest!

```
char memory_pool[100000000]; // 100 million bytes
char *next_pool_addr = memory_pool;

void *malloc(int nbytes)
{
    if (next_pool_addr + nbytes <= memory_pool + sizeof(memory_pool)) {
        void *block_addr = next_pool_addr;
        next_pool_addr += nbytes;
        return block_addr;
        }
    else
        return 0;
}
void free(void *block_addr) { }
```

My **malloc** is *O(1)* and very easy to understand, but does it have any limitations?

There's no provision to actually free memory!

Under what circumstances would it work?

Total memory "throughput" is less than 100 million bytes. (Easily raised, too!)

Example: **fromTo**

Problem:
Write a function **int *fromTo(int from, int to)** that returns a pointer to an allocated array of **int** values ranging from **from** through **to**, inclusive. Assume that **from** <= **to**.

Example of use, with **print_ints** from **a8**:

```
int *a3 = fromTo(1,3);
print_ints(a3, 3); // Output:{1,2,3}
```

Solution:

```
int *fromTo(int from, int to)  // fromTo.c
{
   int *buf = malloc((to - from + 1) * sizeof(*buf));
   int *p = buf;

   while (from <= to)
      *p++ = from++;

   return buf;
}
```

Why **sizeof(*buf)**?
It's another way to reduce the chance of a type mismatch with the **sizeof** expression.

Nov 9 update: I'm seeing the above idea lead to trouble, so let's call it a BAD idea. Just use **... * sizeof(int).**

fromTo, continued

Let's allocate three arrays, printing each, and then clean up.

```
int *a3 = fromTo(1,3);
print_ints(a3, 3);
    // Output: {1,2,3}

int *a3a = fromTo(-3,2);
print_ints(a3a, 6);
    // Output: {-3,-2,-1,0,1,2}

int *a1 = fromTo(100,100);
print_ints(a1-1, 3);
    // Output: {0,100,164}
    // (Note that first and last values are indeterminate.)

int *allocs[] = {a3, a3a, a1, 0};
int **ap = allocs;

while (*ap)
  free(*ap++);
```

fromTo, continued

Are there any issues with the following statement?

```
print_ints(fromTo(-1,1), 3);
```

It's a memory leak! Why?

We've "lost" the address of the allocated block!

Would it be good for **print_ints** to free the memory?

That's not behavior an experienced C programmer would expect.

And, then we couldn't use **print_ints** to print a local (non-allocated) array!

Another angle: What if **free** returned the address it was given, so calls could be nested, like the following?

```
print_ints(free(fromTo(-1,1)), 3);
```

No good! Why?

The above is a "used after freed" error.

fromTo, continued

a8's **print_ints** expects the number of values to print as a second argument:

```
void print_ints(int *first, int n)
```

Problem:

Write a function **print_all** that specifically works with the address returned by **fromTo**. It prints the values in the newly allocated block, and does not require a count.

Example:

```
print_all(fromTo(1,3));     // Output: {1,2,3}

print_all(fromTo(-3,2));    // Output: {-3,-2,-1,0,1,2}
```

Impossible! Why?

There's no sentinel value and there's no way to know the size of the allocated block.

concat

Consider a function **char *concat(char *s1, char *s2)** that returns a dynamically allocated string that contains the concatenation of **s1** and **s2**.

Usage:

```
char *s1 = "concat";
char *s2 = concat(s1, "enate");

puts(s2);
    // Output: concatenate
char *s3 = concat(s2, s2);
puts(s3);
    // Output: concatenateconcatenate

free(s1);
free(s2);
free(s3);
```

Any bugs above?

Yes! **free(s1)** is freeing a string literal!

concat, continued

At hand:

char *concat(char *s1, char *s2) returns a dynamically allocated string that contains the concatenation of **s1** and **s2**.

```
char *s2 = concat("just", " testing");
```

Problem: Write it!

Solution:

```
char *concat(char *s1, char *s2)
{
   int len1 = strlen(s1);
   char *result = malloc((len1+strlen(s2)+1)*sizeof(char));

   strcpy(result, s1);
   strcpy(result + len1, s2);

   return result;
}
```

How about **strcat(result, s2)** instead?

As is, two passes are made over the strings: One to get their lengths, and a second to copy their characters into the allocated memory. Can one pass be eliminated?

realloc

We can "change" the size of a block of allocated memory with **realloc**:

```
void *realloc(void *ptr, size_t size);
```

ptr is the address of a previously allocated block. There are two possible results for **realloc**:

1. The allocator simply adjusts its notion of the size of the block. In this case, **realloc** returns **ptr**.

2. The allocator allocates a new block, copies data from the block referenced by **ptr** into the new block, does **free(ptr)**, and returns the address of the new block.

Imagine an allocator that has two "pools" of blocks: 1000-byte blocks, and variable-sized blocks.

- Resizing a 100-byte block to a 300-byte block might only change an **int** named **size** from 100 to 300. (First case above.)

- Resizing a 100-byte block to a 1001-byte block would cause a variable-size block to be allocated. Data would then be copied into it from the 100-byte block, and the 100-byte block would be freed. (Second case above.)

realloc, continued

Here's an example that shows the mechanics:

```
int main()
{
    char *p = malloc(SZ1);
    paddr(p);
    strcpy(p, "testing");

    char *rp = realloc(p, SZ2);
    paddr(rp);

    puts(rp);
    free(rp);
}
```

Resizing from 10 to 2000 <u>does not</u> cause relocation but going from 10 to 200,000 <u>does</u> cause relocation:

```
% gcc -DSZ1=10 -DSZ2=2000 realloc1.c && a.out
p = 28192784
rp = 28192784
testing

% gcc -DSZ1=10 -DSZ2=200000 realloc1.c && a.out
p = 8085520
rp = 139825191907344
testing
```

realloc, continued

Consider the problem of sorting a number of floating-point values read from standard input. We want to read all the values into memory but we don't know how many values there may be.

Here's pseudo-code for an approach using **realloc**:

```
N = 100
vals = malloc(N*sizeof(double))
while I can read a value
    store the value in the next position in vals
    if vals is full
        N *= 2
        newvals = realloc(vals, N*sizeof(double))
        ~~copy from vals into newvals~~ [brain cramp--realloc does the copy!]
        vals = newvals
```

The code above starts with space for 100 values. Whenever space runs out, it asks for twice as much space as the last allocation.

Let's implement it!

realloc, continued

```
int main() // realloc2.c
{
   int slots = 100;
   double *vals = malloc(slots * sizeof(double));
   double *next = vals, *end = vals + slots;

   while (scanf("%lg", next) == 1) {  // Note: not &next!
      next++;
      if (next == end) {
         slots *= 2;
         int pos = next - vals;
         vals = realloc(vals, slots * sizeof(double));
         next = vals + pos;
         end = vals + slots;
         }
      }

   int numvals = next - vals;
   printf("%d values loaded:\n", numvals);
   for (int i = 0; i < numvals; i++)
      printf("vals[%d] = %g\n", i, vals[i]);

   free(vals);
}
```

Exercise: Using **slots = 1**,

1. Step through it with **gdb**. (Use **run < nums** inside **gdb**.)
2. Work through it with pencil and paper

Pay attention to **vals**, **next**, and **end**.

calloc

Here's what the C11 standard shows for **calloc**:

7.22.3.2 The calloc function

Synopsis

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description

The **calloc** function allocates space for an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all bits zero.

Let's get 25 **int** zeroes:

```
int *vals = calloc(25, sizeof(int));
```

Does the following code produce 25 **double** values of **0.0**?

```
double *dvals = calloc(25, sizeof(double));
```

Fine point: C11 does not require that a **double** with all bits zero equals **0.0**! The above code is not portable.

Similarly, a pointer with all bits zero might not be **== 0**. (!)

valgrind

(say "val-grinned")

What is Valgrind?

valgrind.org says:

> Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

We'll be using Valgrind's Memcheck tool to help us find memory management errors.

There are other such tools, such as BoundsChecker and Purify, but Valgrind is FOSS.

Running **valgrind**

Valgrind is dead-simple to use:

```
% cat -n valgrind1.c
   1  #include <stdlib.h>
   2  int main()
   3  {
   4      char *p = malloc(100);
   5      p[100] = 'x';
   6  }

% gcc valgrind1.c

% valgrind --leak-check=full a.out
==8524== Memcheck, a memory error detector
==8524== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==8524== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==8524== Command: a.out
==8524==
==8524== Invalid write of size 1
==8524==    at 0x400512: main (valgrind1.c:5)
==8524==  Address 0x51f00a4 is 0 bytes after a block of size 100 alloc'd
==8524==    at 0x4C2B6CD: malloc
==8524==    by 0x400505: main (valgrind1.c:4)
...and that's not all...
```

1. **8524** is the process id. (Not useful for us.)
2. **p[100] = 'x'** is noted as "**Invalid write of size 1**" because it's outside of an allocated block.
3. **valgrind** goes on to say that the write was just beyond a block allocated at line 4.

The memory managed by **malloc** is considered to be "the heap".

```
% cat -n valgrind1.c
  1  #include <stdlib.h>
  2  int main()
  3  {
  4      char *p = malloc(100);
  5      p[100] = 'x';
  6  }
```

[**valgrind** output, continued]

```
==8524==
==8524==
==8524== HEAP SUMMARY:
==8524==     in use at exit: 100 bytes in 1 blocks
==8524==   total heap usage: 1 allocs, 0 frees, 100 bytes allocated
==8524==
==8524== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8524==    at 0x4C2B6CD: malloc
==8524==    by 0x400505: main (valgrind1.c:4)
==8524==
==8524== LEAK SUMMARY:
==8524==    definitely lost: 100 bytes in 1 blocks
==8524==    indirectly lost: 0 bytes in 0 blocks
==8524==      possibly lost: 0 bytes in 0 blocks
==8524==    still reachable: 0 bytes in 0 blocks
==8524==         suppressed: 0 bytes in 0 blocks
```

Leaked memory

Shows where the leaked memory was allocated.

"Used after freed" with **valgrind**

```
% valgrind --leak-check=full a.out
==2002== Command: a.out
==2002==
==2002== Invalid read of size 4
==2002==    at 0x4005CE: main (valgrind2.c:16)
==2002==  Address 0x51f0040 is 0 bytes inside a
          block of size 40 free'd
==2002==    at 0x4C2A82E: free ...
==2002==    by 0x40059B: f (valgrind2.c:7)
==2002==    by 0x4005C9: main (valgrind2.c:15)
==2002==
*p is 7
==2002==
==2002== HEAP SUMMARY:
==2002==     in use at exit: 0 bytes in 0 blocks
==2002==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==2002==
==2002== All heap blocks were freed -- no leaks are possible
```

```
5  void f(int *p)
6  {
7    free(p);
8  }
9
10  int main()
11  {
12    int *p =
13     malloc(10*sizeof(int));
14    *p = 7;
15    f(p);
16    iprint(*p);
17  }
```

Note "stack trace"!

No memory leaks.

Let's check **realloc2.c**

Let's use **valgrind** to check **realloc2.c**, the **realloc** example from slide 382.

We'll use valgrind's **-q** option to suppress output unless there are errors.

```
% gcc realloc2.c
% seq 3 | valgrind -q --leak-check=full a.out
3 values loaded:
vals[0] = 1
vals[1] = 2
vals[2] = 3
%
```

Note that we're piping into **valgrind**. **valgrind** arranges for its standard input to be standard input for the program it runs, which is **a.out** in this case.

If **a.out** required arguments, we'd just add them to command line. Let's imagine a **--reverse** option for **realloc2.c**:

```
% seq 3 | valgrind -q --leak-check=full a.out --reverse
...
```

Let's introduce a couple of errors in **realloc2.c**:

```
...
for (int i = 0; i <= numvals; i++) // should be <
  printf("vals[%d] = %g\n", i, vals[i]);
   //free(vals);  // skip the free!
}
```

Execution:

```
% gcc realloc2.c
% seq 3 | valgrind ...
3 values loaded:
vals[0] = 1
vals[1] = 2
vals[2] = 3
==29134== Conditional jump or move depends on uninitialised value(s)
==29134==    at 0x4E80043: __printf_fp (printf_fp.c:406)
==29134==    by 0x4E7D116: vfprintf (vfprintf.c:1596)
==29134==    by 0x4E85298: printf (printf.c:35)
==29134==    by 0x400720: main (realloc2.c:24)
==29134==
```

<u>...several more of the above, but at different lines in printf_fp.c...</u>

```
vals[3] = 0
==29134== 32 bytes in 1 blocks are definitely lost in loss record 1 of 1
==29134==    at 0x4C2B7B2: realloc ...
==29134==    by 0x400678: main (realloc2.c:15)
==29134==
```

Let's check **/usr/bin/tac**

We can run **valgrind** on any executable, although full line-number information might not be available.

```
% date | valgrind --leak-check=full /usr/bin/tac
==22441== Command: /usr/bin/tac
==22441==
Wed Nov  4 01:16:36 MST 2015
==22441==
==22441== HEAP SUMMARY:
==22441==     in use at exit: 16,971 bytes in 3 blocks
==22441==   total heap usage: 166 allocs, 163 frees, 36,821 bytes allocated
==22441==
==22441== 16,388 bytes in 1 blocks are possibly lost in loss record 3 of 3
==22441==    at 0x4C2B6CD: malloc ...
==22441==    by 0x404338: ??? (in /usr/bin/tac)
==22441==    by 0x401854: ??? (in /usr/bin/tac)
==22441==    by 0x4E5376C: (below main) (libc-start.c:226)
==22441==
==22441== LEAK SUMMARY:
==22441==    definitely lost: 0 bytes in 0 blocks
==22441==    indirectly lost: 0 bytes in 0 blocks
==22441==      possibly lost: 16,388 bytes in 1 blocks
==22441==    still reachable: 583 bytes in 2 blocks
==22441==         suppressed: 0 bytes in 0 blocks
...
```

valgrind doesn't catch non-heap errors

Valgrind's Memcheck tool doesn't catch non-heap errors.

```
% gcc valgrind3.c && valgrind --leak-check=full a.out
==2196== Memcheck, a memory error detector
...
==2196== Command: a.out
==2196==
==2196==
==2196== HEAP SUMMARY:
==2196==     in use at exit: 0 bytes in 0 blocks
==2196==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==2196==
==2196== All heap blocks were freed -- no leaks are possible
==2196==
==2196== For counts of detected and suppressed errors, rerun with: -v
==2196== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2
from 2)
```

```
% cat valgrind3.c
int main()
{
   int a[3];

   a[3] = a[-1];
}
```

There is an experimental Valgrind tool, SGcheck, that looks for overruns in stack and global objects. It has issues, but feel free to try it. Example:

```
% gcc valgrind3.c && valgrind --tool=exp-sgcheck a.out
```

Use **valgrind**!

On most assignment problems that require memory allocation your solutions will be expected to be "**valgrind** clean"—no errors detected by **valgrind**.

Here's a handy alias for your **.bashrc**:

```
alias vg="valgrind --leak-check=full"
```

Valgrind throws a lot of technology at the problem of manual memory management but even rudimentary tools can be useful.

A portion of assignments 10 and 11 will be implementing a simple memory allocator that detects leaks and bad frees.

Loose end: `sprintf` et al.

The problem

It's possible to build up a string using a series of **strcpy** and **strcat** calls, other routines, and ad-hoc code but it's very clumsy and error prone.

Imagine a **make_name** routine that assembles a name from first/middle/last names:

```
char result[100];
make_name("John", "Quincy", "Adams", result);
puts(result);  // Output: John Q. Adams
```

Implementation: (ouch!)

```
void make_name(char *first, char *middle, char *last, char *result)
{
    strcpy(result, first);
    strcat(result, " ");

    char initial[] = "X. ";
    initial[0] = middle[0];
    strcat(result, initial);

    strcat(result, last);
}
```

Solution: **sprintf**

The **sprintf** routine is like **printf** but instead of writing to standard output, **sprintf** writes a string into memory.

Here is its prototype:

```
int sprintf(char *result, const char *format, ...);
```

Here is **make_name** with **sprintf**:

```
void make_name(char *first, char *middle, char *last, char *result)
{
    sprintf(result, "%s %c. %s", first, middle[0], last);
}
```

All the formatting capabilities of **printf** are available in **sprintf**.

What's a hazard with **sprintf**?

The caller must ensure the target buffer, **result**, has sufficient room.

Two safe alternatives, but with complications:

- **asprintf**—print to allocated string
- **snprintf**—like **sprintf** but with buffer length specified

sprintf, continued

Imagine a routine that creates a string with a comma-separated list of array values:

```
int a[] = {5, -10, 30, 7};
char buf[100];
ints_to_str(a, 4, buf);  // buf now holds "5, -10, 30, 7"
```

A solution with **sprintf**:

```
void ints_to_str(int *vals, int length, char *result)
{
   if (length == 0) {
      *result = 0;
      return;
      }
   for (int *p = vals; p < vals + length; p++)
      result += sprintf(result, "%d, ", *p);
   result[-2] = 0; // eliminate trailing ", "  (hack or "technique"?)
}
```

The code takes advantage of the that fact that **sprintf** returns the number of characters inserted into the target buffer, not counting the terminating zero.

Structures

Structure basics

Structures provide a way to aggregate values as a collection of named elements.

Structure *tags* are declared with the **struct** keyword. Example:

```
struct Rectangle {
    int width;
    int height;
    };
```

Rectangle is said to have two *members*: **width** and **height**.

The declaration of a structure must appear before it can be used. Structures are typically declared outside of functions.

Note the family resemblance to Java. **struct**s in C++ look the same.

Unlike Java, there is no way to associate functions with **Rectangle**. The best we can do in C is to name functions descriptively, such as **area_of_Rectangle** or **printRectangle**.

IMPORTANT: Don't forget the semicolon at the end of a **struct** declaration!

Structure basics, continued

For reference:

```
struct Rectangle {
    int width;
    int height;
    };
```

The following declaration creates a **Rectangle**. Note that **struct** is required.

```
struct Rectangle r;
```

This reserves space for a **Rectangle** named **r**, just like **int x;** reserves space for an **int** named **x**.

sizeof can be applied to an instance of **Rectangle** or the tag:

```
sizeof(r)                   // 8
sizeof(struct Rectangle)    // 8 (struct is required)
```

Structure basics, continued

For reference:

```
struct Rectangle {
    int width;
    int height;
    };

struct Rectangle r;
```

One way to reference members is with the "dot" operator, which is sometimes called *member selection*.

```
r.width = 10;
r.height = 20;

int area = r.width * r.height;
```

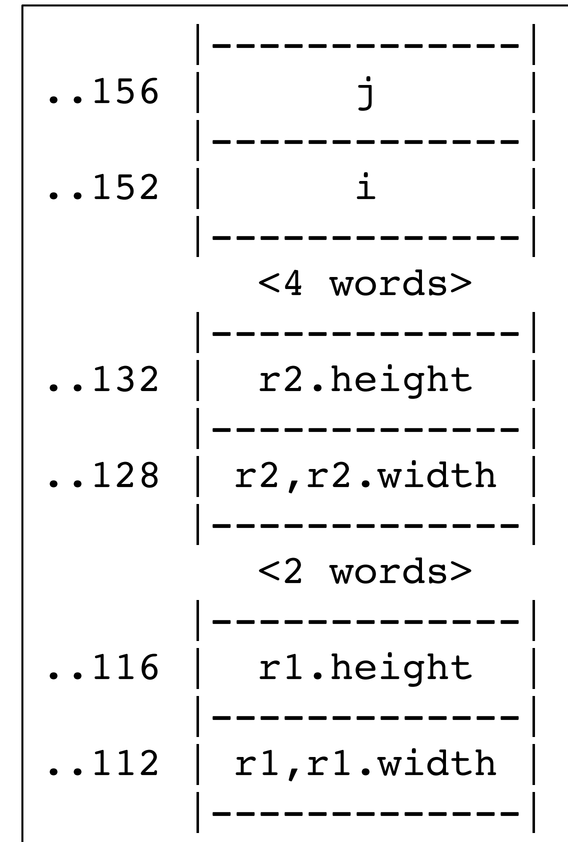
Unlike Java, C has no notion of access control—all members are "public".

In C, there's no simple way to hide the representation of an object. Code all over a system can come to rely on **struct** members that we might later want to change.

Structure basics, continued

It is possible to get the address of a structure, and addresses of individual members of a structure.

```
int main()
{
    int i, j;
    struct Rectangle r1, r2;

    printf("&r1 = %lu\n", &r1);
    printf("&r1.width = %lu\n", &r1.width);
    ...lots more...
```



Structure members appear in memory in the order they are declared. The address of a structure and the address of its first element are the same.

`r1` and `r2` are local variables. They have automatic storage duration and reside on the stack along with other locals.

Like other objects with automatic storage duration, <u>the members of `r1` and `r2` are not initialized.</u>

Structure basics, continued

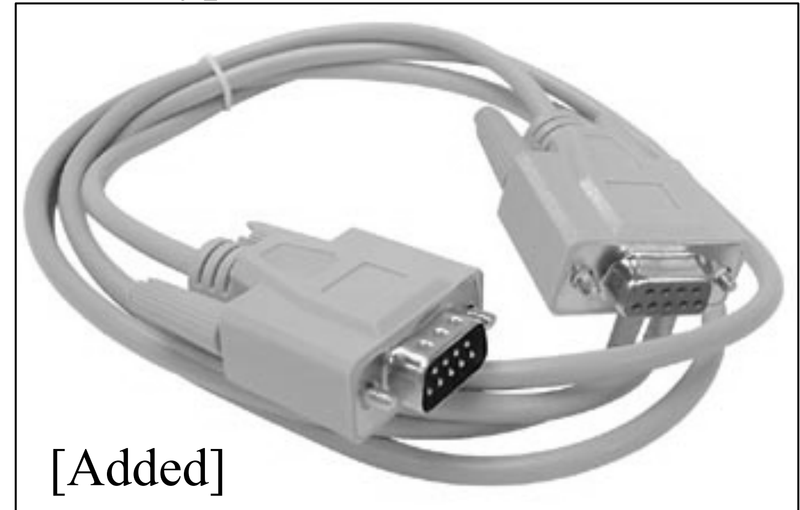
We can have an array of structures. Member selection can be applied to elements of the array.

```
struct Rectangle { int width; int height; };
int main() // struct1a.c
{
    struct Rectangle rects[5];
    int nrects = sizeof(rects) / sizeof(rects[0]);

   for (int i = 0, w = 3, h = 4; i < nrects; i++, w += 2, h += 3) {
        rects[i].width = w;
        rects[i].height = h;
        }

    int sum_of_areas = 0;
    for (int i = 0; i < nrects; i++)
        sum_of_areas += rects[i].width * rects[i].height;

    printf("Sum of areas: %d\n", sum_of_areas);
}
```

Structure basics, continued

Structures can contain elements of any combination of types. Consider a structure to represent a serial communication cable:

```
struct SerialCable {
    double length;
    int conductors;
    char ends[2]; // 'M' or 'F'
    char *manufacturer;
    };
```


[Added]

To create and initialize a **SerialCable** one might do this:

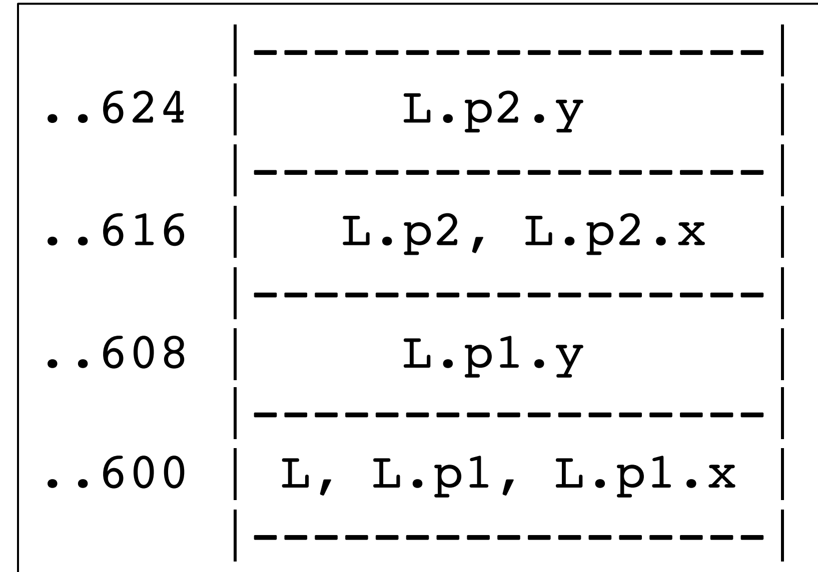
```
struct SerialCable sc;
sc.length = 5.5;
sc.conductors = 9;
sc.ends[0] = 'M';
sc.ends[1] = 'F';
sc.manufacturer = "Cables 'R Us";
```

Although an instance of **SerialCable** holds only 22 bytes of data, **sizeof(struct SerialCable)** produces 32! The extra space is padding to satisfy data alignment requirements.

Nested structures

Structures can contain other structures:

```
struct Point {
    double x;
    double y;
    };

struct Line {
    struct Point p1, p2;
    };

struct Line L;
```



Note that each instance of **Line** physically contains two instances of **Point** that each physically contain two **double** values.

sizeof(struct Point) is 16, **sizeof(struct Line)** is 32, and **sizeof(L)** is 32.

A slope calculation:

```
L.p1.x = 1;  L.p1.y = 1;
L.p2.x = 3;  L.p2.y = 4;
double slope = (L.p2.y - L.p1.y) / (L.p2.x - L.p1.x);
```

L, an instance of **struct Rectangle** is a single memory object. Imagine the Java analog for **Line**. How many objects would be required for an instance of it?

Nested structures, continued

For reference:

```
struct Point {
    double x;
    double y;
    };

struct Line {
    struct Point p1, p2;
    };
```

Here is an alternative definition for **Line**:

```
struct Line {
    struct Point p[2];
    };
```

A slope calculation:

```
struct Line L;

L.p[0].x = 1; L.p[0].y = 1;
L.p[1].x = 3; L.p[1].y = 4;

double slope = (L.p[1].y - L.p[0].y) / (L.p[1].x - L.p[0].x);
```

Structure initialization

C's structure initialization syntax is very similar to the array initialization syntax.

For reference:

```
struct SerialCable {
    double length;
    int conductors;
    char ends[2];
    char *manufacturer;
    };
```

An instance can be defined and initialized like this:

```
struct SerialCable cable1 = { 1.5, 3, {'M', 'M'}, "Joe's Cables" };
```

An alternative:

```
struct SerialCable cable2 = { 2.5, 9, 'M', 'F', "Joe's Cables" };
```

If an initializer is omitted, the corresponding member is initialized with a *type-appropriate zero*.

```
struct SerialCable cable3 = { 5.0, 9 };
```

is equivalent to

```
struct SerialCable cable3 = { 5.0, 9, {0, 0}, 0 };
```

Structure initialization, continued

An array of structures can be initialized with a series of initializers:

```
struct SerialCable cables[] = {
    { 1.5, 3, {'M', 'M'}, "Joe's Cables" },
    { 2.5, 9, {'M', 'F'}, "Cables 'R Us" },
    { 4.0, 3, {'F', 'M'}, "connectem.com" }
    };
```

```
struct SerialCable {
        double length;
        int conductors;
        char ends[2];
        char *manufacturer;
        };
```

The grouping braces can omitted, albeit with a loss of clarity:

```
struct SerialCable cables[] = {
    1.5, 3, 'M', 'M', "Joe's Cables", 2.5, 9, 'M', 'F',
    "Cables 'R Us", 4.0, 3, 'F', 'M', "connectem.com"
    };
```

To create an array of **SerialCables** with only the length initialized in each, we can do this:

```
struct SerialCable cables[] = {{2.2},{1.3},{7},{4},{15}};
```

All members aside from length are zeroed.

Structure initialization, continued

For reference:

```
struct SerialCable {
    double length;
    int conductors;
    char ends[2];
    char *manufacturer;
    };
```

C11 supports *designated initializers*, which allow members to be initialized by name (and more).

Here's a simple example:

```
struct SerialCable cables[] = {
  { .conductors = 3,
    .length = 2.0,  // Note difference in order vs. struct
    .manufacturer = "Kinky Kables" },
  { .ends = 'M' }
  };
```

Structure initialization, continued

Instances of Line can be initialized like this:

```
struct Line L1 = {{1, 2}, {3, 4}};
struct Line L2 = {1, 1, -1, -1};
struct Line L3 = {{}, 5, 7};
struct Line a[] = {{1,2}, {}, {3,4}};
```

```
struct Point { double x, y; };
struct Line {
    struct Point p1, p2;
    };
```

One way to explore structure initialization is to print values with gdb:

```
(gdb) p L1
$1 = {p1 = {x = 1, y = 2}, p2 = {x = 3, y = 4}}
(gdb) p L2
$2 = {p1 = {x = 1, y = 1}, p2 = {x = -1, y = -1}}
(gdb) p L3
$3 = {p1 = {x = 0, y = 0}, p2 = {x = 5, y = 7}}
(gdb) p a
$4 = {{p1 = {x = 1, y = 2}, p2 = {x = 0, y = 0}},
    {p1 = {x = 0, y = 0}, p2 = { x = 0, y = 0}},
    {p1 = {x = 3, y = 4}, p2 = {x = 0, y = 0}}}
```

The full set of rules for structure initialization is complex and the designated initializer rules, which apply to simple arrays, too, are even more complex.

Structures as values

It is possible to treat an entire structure as a value. The contents of the structure are simply copied as a sequence of bytes. Example:

```
struct Point { double x, y; };
void print_Point(char *label, struct Point pt);  // Note type of pt
int main()  // struct6a.c
{
   struct Point p1 = {3.1, 4.2}, p2;

   p2 = p1;          // copies sizeof(p1) bytes from &p1 to &p2
   p2.x *= 2.0;   // p1 is unchanged

   print_point("p1", p1);  // pushes sizeof(p1) bytes onto stack
   print_point("p2", p2);  // ditto for p2
}

void print_point(char *label, struct Point pt) // call by value for pt
{
   printf("%s: (%g, %g)\n", label, pt.x, pt.y);
}
```

```
Output:
   p1: (3.1, 4.2)
   p2: (6.2, 4.2)
```

Structures as values, continued

```
struct Point translate(struct Point p, double xdelta, double ydelta)
{
    struct Point newPoint;

    newPoint.x = p.x + xdelta;
    newPoint.y = p.y + ydelta;

    return newPoint;
}

double slope(struct Line L)
{
    return (L.p2.y - L.p1.y) / (L.p2.x - L.p1.x);
}

int main() // struct6.c
{
    struct Line A = {{0,0},{2,4}};

    printf("slope of A = %g\n", slope(A));

    A.p1 = translate(A.p1, 0, 10);

    printf("new slope of A = %g\n", slope(A));
}
```

For reference:

```
    struct Point {
        double x, y;
        };

    struct Line {
        struct Point p1, p2;
        };
```

Which lines treat a structure as a value?

What effect would be produced by the statement **translate(A.p2, 3, 4);**?

No effect. A point would be created and returned but never used.

Pointers to structures

A pointer variable that can reference an instance of **Rectangle** is declared like this:

```
struct Rectangle *rp;
```

Let's make a **Rectangle** named **r** and point **rp** at it:

```
struct Rectangle r;
rp = &r;
```

We can use the **->** operator to reference a member of a pointed-to structure:

```
rp->width = 5;
rp->height = 7;

int area = rp->width * rp->height;
```

```
r.width = 5;
r.height = 7;

int area = r.width * r.height
```

This operator, like the **.** ("dot") operator, is sometimes called *member selection* or *member access*, but is usually read as "*pointing to*". Like "dot", **->** has very high precedence.

A precise reading of **rp->width** is "the **width** member of the **Rectangle** that **rp** points to".

Pointers to structures, continued

For reference:

```
struct Rectangle r;
struct Rectangle *rp;
rp = &r;
```

We can apply * to `rp` to produce an L-value that specifies `r`.

Note the types:

```
rp   is  pointer to struct Rectangle
*rp  is  struct Rectangle
```

Let's assign the contents of `r` (pointed to by `rp`) to `r2`:

```
struct Rectangle r2;
r2 = *rp;
```

Pointers to structures, continued

Recall that we can take the address of member referenced with "dot":

```
int *p1 = &r.width;
```

Like "dot", the -> operator produces an L-value. This expression is valid:

```
int *p2 = &rp->width;   // Precedence: &(rp->width)
```

It says "Assign to p2 the address of the width member of the Rectangle pointed to by rp.

To swap the width and height of a Rectangle we can do this:

```
swap(&rp->width, &rp->height)
```

Sidebar: An equivalence with . and ->

The general form of the . (dot) operator is:

L-value . member-name

Given **struct Rectangle** **r**, **r** is an L-value: it specifies a particular **Rectangle** object in memory.

If the type of **rp** is **struct Rectangle ***, then ***rp** yields an L-value of type **struct Rectangle**.

Thus,

```
(*rp).width
```

is completely equivalent to

```
rp->width
```

K&R 2e describes **p->m** as a shorthand for **(*p).m**

Pointers to structures, continued

A pointer can iterate over an array of structures. Example:

```
struct Rectangle rects[5];

//...populate rects with widths and heights...

int sum_of_areas = 0;
int nrects = sizeof(rects)/sizeof(*rp);
for (struct Rectangle *rp = rects;  rp < rects + nrects;  rp++)
    sum_of_areas += rp->width * rp->height;

printf("Sum of areas: %d\n", sum_of_areas);
```

Notes:

rp++ adds **sizeof(struct Rectangle)** to **rp**, advancing it to the next element.

rects + nrects is equivalent to **&rects[nrects]**.

The **for** loop continues until **rp** reaches **rects + nrects**, which would be the address of the sixth rectangle, which is non-existent.

Pointers to structures, continued

An alternative to passing a structure by value is to pass a pointer to a structure.

```
void print_Point(char *label, struct Point *pt)
{
    printf("%s: (%g, %g)\n", label, pt->x, pt->y);
}
```

Usage:

```
struct Point p1 = {3.1, 4.2};
print_Point("p1: ", &p1);
```

Contrast with the value-passing version shown earlier:

```
void print_Point(char *label, struct Point pt)
{
    printf("%s: (%g, %g)\n", label, pt.x, pt.y);
}
```

Usage:

```
struct Point p1 = {3.1, 4.2};
print_Point("p1: ", &p1);
```

What's a performance consideration when deciding whether to have a function take a structure by value vs. passing a pointer?

If a structure is large, passing a pointer avoids copying it onto the stack.

Structure initialization with a function

It's common to have functions that initialize a structure using values passed as arguments.

```
void init_cable(struct SerialCable *scp,
   double len, int wires, char end1, char end2, char *mfr)
{
   scp->length = len;
   scp->conductors = wires;
   scp->ends[0] = end1; // Could also be *scp->ends = end1;
   scp->ends[1] = end2;
   scp->manufacturer = strdup(mfr); // Copies to malloc'd memory
}
```

Usage:

```
struct SerialCable sc;
init_cable(&sc, 2.0, 10, 'M', 'F', "connectem.com");

struct SerialCable *p = malloc(sizeof(struct SerialCable));
init_cable(p, 3.5, 20, 'F', 'M', "Point-to-Point, Inc.");
```

Would it be better to put the **malloc** inside **init_cable**?

How about a flag to indicate that **init_cable** should also allocate memory?

An approximation of a Java constructor

A function that allocates memory for an object, initializes it based on parameters, and then returns a pointer to it is a rough approximation of a Java constructor.

```
struct Line *make_line(double x1, double y1, double x2, double y2)
{
    struct Line *lp = malloc(sizeof(struct Line));
    lp->p1.x = x1;   lp->p1.y = y1;
    lp->p2.x = x2;   lp->p2.y = y2;
    return lp;
}
```

Usage:

```
struct Line *lp = make_line(1, 2, 3, 4);
```

Contrast with Java:

```
Line line = new Line(1, 2, 3, 4);
```

make_line is fairly close to a Java constructor. It creates and initializes an object in the heap and returns a pointer. A pointer to a structure is essentially equivalent to a reference in Java.

Don't return a pointer to a local structure!

Just like returning a pointer to a local array is a mistake, so is returning a pointer to a structure that's a local variable.

Here's an example of a mistake:

```
struct Line *make_line(double x1, double y1, double x2, double y2)
{
   struct Line L = { {x1, y1}, {x2, y2} };

   return &L;  // DON'T DO THIS!!
}
```

The lifetime of **L** ends when **make_line** returns but the returned pointer will still reference the memory where **L** resided during the lifetime of **make_line**.

It can be said that **make_line** returns a *dangling pointer*—it's a pointer that was once valid but no longer is.

gcc does produce a warning: **function returns address of local variable**

Example: **getpwent()**

The **getpwent()** library function is used to iterate through **/etc/passwd** entries. Here's the prototype that **man getpwent** shows:

```
struct passwd *getpwent(void);
```

The man page goes on to show this:

The **passwd** structure is defined in **<pwd.h>** as follows:

```
struct passwd {
    char   *pw_name;       /* username */
    char   *pw_passwd;     /* user password */
    uid_t   pw_uid;        /* user ID */
    gid_t   pw_gid;        /* group ID */
    char   *pw_gecos;      /* user information */
    char   *pw_dir;        /* home directory */
    char   *pw_shell;      /* shell program */
};
```

uid_t and **gid_t** are **typedef**s (type aliases) for **unsigned int**s.

getpwent(), continued

The first call to **getpwent()** returns a pointer to a structure populated with data from the first line in **/etc/passwd**. Let's see what the first line is:

```
% head -1 /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Here's a program that simply reads and prints the first **/etc/passwd** entry:

```
#include <stdio.h>
#define __USE_SVID  // needed to "see" getpwent() prototype
#include <pwd.h>
int main() //getpwent1.c
{
   struct passwd *ep = getpwent();
   printf("username: %s, real name: %s, home: %s, shell: %s\n",
      ep->pw_name, ep->pw_gecos, ep->pw_dir, ep->pw_shell);
}
```

Execution:

```
% gcc getpwent1.c && a.out
username: root, real name: root, home: /root, shell: /bin/bash
```

getpwent(), continued

Let's look for "real name" (pw_gecos) entries that contain a string specified on the command line. Example:

```
% getpwent2 Mitchell
[150] username: whm, real name: William H. Mitchell
[286] username: steve, real name:  Steven Maury Mitchell
[764] username: mjmitchell, real name:  Matthew John Mitchell
...lots more...
```

Implementation:

```
int main(int argc, char **argv) //getpwent2.c
{
    struct passwd *ep;
    int entry_num = 0;
    while ((ep = getpwent())) {
        entry_num++;
        if (strstr(ep->pw_gecos, argv[1])) {
            printf("[%d] username: %s, real name: %s\n",
                entry_num, ep->pw_name, ep->pw_gecos);
        }
    }
}
```

getpwent(), continued

Recall that first example with **getpwent()**:

```
int main() //getpwent1.c
{
   struct passwd *ep = getpwent();
   printf("username: %s, real name: %s, home: %s, shell: %s\n",
      ep->pw_name, ep->pw_gecos, ep->pw_dir, ep->pw_shell);
}
```

Where is the instance of **struct passwd** that **ep** is pointing to? What is its lifetime?

From the man page:

> The return value may point to a static area, and may be overwritten by subsequent calls to **getpwent()**, **getpwnam(3)**, or **getpwuid(3)**. (Do not pass the returned pointer to **free(3)**.)

getpwent(), continued

Here's a skeletal **getpwent()** implementation:

```
struct passwd *getpwent(void)
{
   static struct passwd result;
   // ...populate result...
   return &result; // OK because it's static!
}
```

Because **result** is declared to be **static**, it survives across calls to **getpwent()**.
Therefore, it's ok to return a pointer to it.

What's another static data value that **getpwent()** is apparently using?

Hint: **while ((ep = getpwent())) { ... }**

Successive calls to **getpwent()** read successive lines from **/etc/passwd**, so **getpwent()** perhaps also has a static variable that holds some sort of file object that tracks the current position in **/etc/passwd**.

What's a pitfall with **getpwent()**?

Data "returned" by a call gets overwritten by the next call.

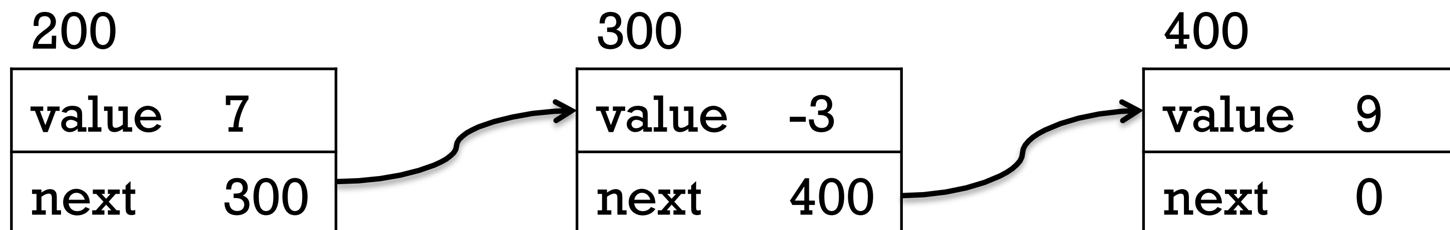
A linked list

C structures are used to implement data structures like linked lists. Here's a **struct** that represents nodes in a singly-linked list of **int**s:

```
struct node {
    int value;
    struct node *next;
    };
```

Note that the declaration is self-referential: the member **next** points to a **node**.

Here's how the sequence 7, -3, 9 might be represented with three nodes:



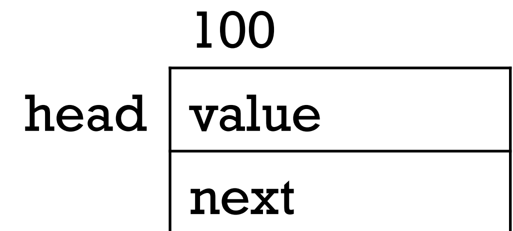
Nodes in a linked structure are almost always in allocated memory, with the possible exception of "head"/"root" nodes in some cases.

A linked list, continued

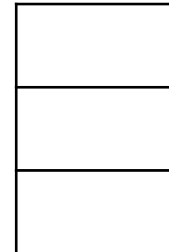
For reference:

```
struct node {
    int value;
    struct node *next;
    };
```

Input: 7, -3, 9



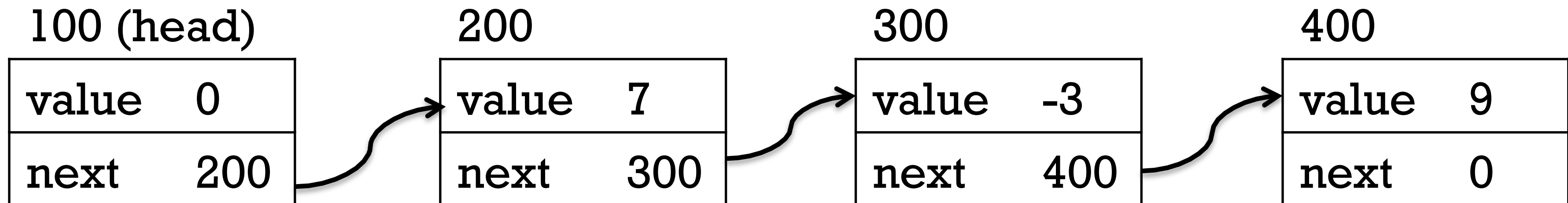
Here's code that reads **int**s from standard input, building a list and then printing it: (**linked1.c**)



```
struct node head = {0, 0};
struct node *last = &head;
int num;

while (scanf("%d", &num) == 1) {
    struct node *new = malloc(sizeof(struct node));
    new->value = num;
    new->next = 0;
    last->next = new;
    last = new;
    }

for (struct node *np = head.next; np; np = np->next)
    printf("%d\n", np->value);
```

A linked list, continued

Problem: Deallocate the memory.



tofree

np

```
struct node *np = head.next;
while (np) {
   struct node *tofree = np;
   np = np->next;
   free(tofree);
   }
```

A sorted linked list

Variation: let's maintain the list in sorted order instead of adding nodes at the end.

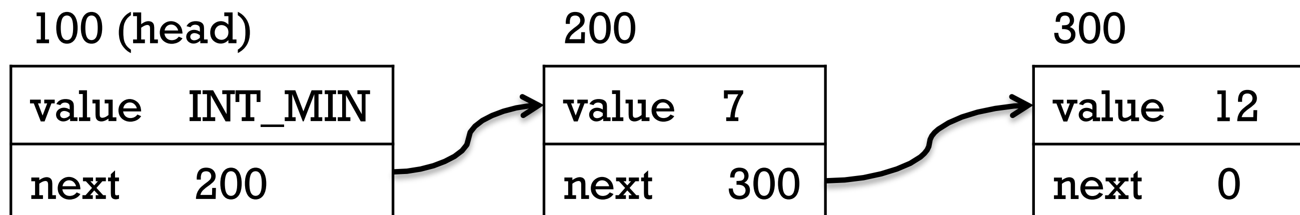
Step 1: Have the **head** node hold a minimum value:

```
struct node head = {INT_MIN, 0};
```

Step 2: Find a node in the list such that **num** is **>=** the node's value and **<=** the next node's value. Or, add the new node at the end. Here's the code:

```
while (scanf("%d", &num) == 1) {
    struct node *new = malloc(sizeof(struct node));
    new->value = num; new->next = 0;

    for (struct node *np = &head; ; np = np->next) {
        if (np->next == 0 ||
            (num >= np->value && num <= np->next->value)) {
            new->next = np->next;
            np->next = new;
            break;
        }
    }
}
```

Input: ~~12~~, ~~7~~, 9, 15



I/O streams and more

I/O streams and FILE

The C library has a family of functions for working with I/O *streams*. Prototypes for the functions are in **<stdio.h>**.

The **fopen()** function opens a file and returns a pointer to a data structure that represents a stream. Here is the prototype:

```
FILE *fopen(const char *path, const char *mode);
```

Let's open a stream to read from the file **lines**:

```
FILE *in = fopen("lines", "r");
```

With **gcc**, **FILE** is defined as a **typedef**:

```
typedef struct _IO_FILE {
   ...lots...
   } FILE;
```

The C11 standard requires that there be a **FILE** type, but the **_IO_FILE** structure is **gcc**-specific. Other implementations of the C library might have a different underlying **struct**, but we can count on **FILE** being present.

fgetc and fputc

There are stream-based counterparts for **getchar** and **putchar**:

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

Here's a simple-minded version of **cp(1)**:

```
int main(int argc, char *argv[]) // cp0.c
{
   FILE *in = fopen(argv[1], "r");
   FILE *out = fopen(argv[2], "w");

   int c;
   while ((c = fgetc(in)) != EOF) {
      fputc(c, out);
      }

   fclose(in);
   fclose(out);
}
```

Usage:

```
% cp0 /etc/passwd pw
% diff pw /etc/passwd
%
```

An error

For reference:

```
int main(int argc, char *argv[]) // cp0.c
{
   FILE *in = fopen(argv[1], "r");
   FILE *out = fopen(argv[2], "w");
   ...
   int c;
   while ((c = fgetc(in)) != EOF) {
      fputc(c, out);
      }
   ...
```

Another copy:

```
% cp0 /etc/password pw
Segmentation fault (core dumped)
```

Why?

If **fopen** fails for any reason, it returns **NULL** (0).

perror—look up an error message

An **fopen** call can fail for any one of several reasons. It is common for failing library routines to set **errno**, a global variable.

void perror(char *s) looks up the error code held in **errno** and prints an explanatory error message, preceded by **s**. Example:

```
int main(int argc, char **argv) // fopen1.c
{
    FILE *fp = fopen(argv[1], argv[2]);

    if (fp != NULL)
        printf("ok!\n");
    else
        perror(argv[1]);
}
```

Usage:

```
% perror1 /etc/passwd w
/etc/passwd: Permission denied

% perror1 not-here r
not-here: No such file or directory

% perror1 not-here w
ok!

% perror1 . w
.: Is a directory
```

stdin, **stdout**, and **stderr**

Three standard streams, of type **FILE ***, are available: **stdin**, **stdout**, and **stderr**.

Here is yet another trivial **cat**:

```
int main()
{
   int c;
   while ((c = fgetc(stdin)) != EOF)
      fputc(c, stdout);
}
```

Note the analogs:

C	Java
stdin	System.in
stdout	System.out
stderr	System.err

fgets and **fputs**

The **fgets** and **fputs** functions are stream-based counterparts for **gets** and **puts**:

```
char *fgets(char *s, int n, FILE *stream);
int fputs(const char *s, FILE *stream);
```

Unlike **gets**, **fgets** includes a buffer length argument. At most **n-1** characters are read into memory starting at **s**. A line read by **fgets** includes the trailing newline; **fputs** does not add a newline to the string being written.

Here's **cat** with **fgets** and **fputs**:

```
int main() // fcat.c
{
   char line[100];
   while (fgets(line, sizeof(line), stdin) != NULL)
      fputs(line, stdout);
}
```

Would **fcat < /bin/ls > myls** likely work?

No. **fputs**, like **puts**, assumes a C string, and stops at a zero byte.

What's the minimum size that **line** can be and still able to handle any text file?

2

fprintf and fscanf

fprintf is just like **printf** but a **FILE *** precedes the format specification. We can use it with **stderr** to produce output on standard error:

```
int main(int argc, char **argv)
{
   if (argc != 3) {
      fprintf(stderr, "Usage: %s from to\n", argv[0]);
      ...
```

Along with **fprintf**, there is **fscanf**. For all practical purposes,

printf(...); and **scanf(...);**

are equivalent to

fprintf(stdout, ...); and **fscanf(stdin, ...);**

Any **FILE *** returned by **fopen(..., "r")** can be used with **fscanf.**

Any **FILE *** returned by **fopen(..., "w")** can be used with **fprintf.**

getline—read a line of any length

C11 doesn't [appear to] provide any function to read an input line of unlimited length but POSIX.1-2008 added **getline()**, orginally a GNU extension.

Here's the prototype:

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

getline reads bytes from **stream** until it reaches a newline and returns the number of bytes read.

If ***lineptr** is zero, the address of a newly allocated block of memory containing the bytes read, including a final newline, is assigned to ***lineptr**.

If ***lineptr** is non-zero, it assumed to be the address of an allocated block of memory to hold the line. If it turns out to be too small, it is **realloc**'d, and ***lineptr** is updated.

The size of the allocated block, which may be greater than the number of bytes read, is assigned to ***n**.

-1 is returned at end-of-file.

getline, continued

This program uses **getline** to fill the array **lines** with pointers to allocated copies of each line read. **lines** is fixed size but the input lines may be of any length.

```
#define _GNU_SOURCE  // needed to "see" getline prototype
#include <stdio.h>
int main() // getline2.c
{
    char *lines[10000], **next = lines;

    ssize_t bytes_read; size_t bytes_allocd;
    char *result = 0;
    while ((bytes_read = getline(&result, &bytes_allocd, stdin)) != EOF) {
        *next++ = result;
        result = 0;  // Zeroing result forces getline to allocate a new
        }            // block each time through the loop.
    ...
}
```

What would happen if we didn't zero **result**?

getline, continued

At hand:

```
ssize_t bytes_read; size_t bytes_allocd;
char *result = 0;
while ((bytes_read = getline(&result, &bytes_allocd, stdin)) != EOF) {
    ...
```

Does **getline** seem over-complicated?

What would be a simplified interface?

```
char *getline(FILE* stream);
```

We gain simplicity but what would we lose?

- If we wanted to "delete" the trailing newline we'd need to use **strlen**.
- If we need to hold only one line at a time, there would be a lot of unnecessary memory throughput by allocating a fresh block for every line rather than letting **getline** reallocate when needed.

It's the usual dilemma: If we're using C for the sake of efficiency, then don't we want this sort of fine-grained control of behavior?

fread and **fwrite**

The duo of **fgets** and **fputs** can't handle data that zero-value bytes. **fgetc** and **fputc** handle arbitrary data but there's overhead in their character-by-character processing.

fread and **fwrite** provide a way to read or write a large amount of data with a single call. Here are the prototypes:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Note the two middle parameters for both: **size** and **nmemb**. **fread** attempts to read **nmemb** objects that are **size** bytes long. It returns the number actually read.

fwrite writes **nmemb** objects that are **size** bytes long. It returns the number actually written.

fread and **fwrite**, continued

Here's a program that tries to read 4096 bytes and then writes as many as it read:

```
int main(int argc, char *args[]) // fread1.c
{
    char buf[4096];
    size_t bytes_read;
    bytes_read = fread(buf, sizeof(char), sizeof(buf), stdin);

    fprintf(stderr, "read %zd bytes\n", bytes_read);
    fwrite(buf, sizeof(char), bytes_read, stdout);
}
```

Execution:

```
% ls -l /tmp/20m x.c
-rw-rw-r-- 1 whm whm 168888897 Nov 13 02:26 /tmp/20m
-rw-r--r-- 1 whm whm       138 Nov 13 00:27 x.c
% fread1 < x.c >x.c.out
read 138 bytes
% fread1 < /tmp/20m > 20m.out
read 4096 bytes
% ls -l 20m.out x.c.out
-rw-rw-r-- 1 whm whm 4096 Nov 13 03:19 20m.out
-rw-rw-r-- 1 whm whm  138 Nov 13 03:19 x.c.out
```

fread and **fwrite**, continued

Here's a simple **cp**:

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

int main(int argc, char *argv[]) // cp1.c
{
    FILE *in = fopen(argv[1], "r");
    FILE *out = fopen(argv[2], "w");

    char buf[4096];
    size_t bytes_read;
    while ((bytes_read = fread(buf, sizeof(char), sizeof(buf), in)) != 0) {
        fwrite(buf, sizeof(char), bytes_read, out);
        }

    fclose(in); fclose(out);
}
```

fread and **fwrite**, continued

Let's compare the performance of **cp1** from the previous slide, with **cp0** from slide 433, which uses **fgetc**/**fputc**.

```
% ls -l /tmp/20m
-rw-rw-r-- 1 whm whm 168888897 Nov 13 02:26 /tmp/20m

% time cp0 /tmp/20m x

real    0m4.564s
user    0m2.620s
sys     0m0.200s

% time cp1 /tmp/20m x

real    0m2.206s
user    0m0.016s
sys     0m0.236s
```

Experiment: Try **cp1** with other buffer sizes, like 1K, and 100K.

Example: **split0.c**

split(1) breaks a file into pieces. Let's write a simple version of it. Usage:

```
% ls -l x.c
-rw-r--r-- 1 whm whm 138 Nov 13 00:27 x.c

% split0 x.c 50

% ls -l x.c.*
-rw-rw-r-- 1 whm whm 50 Nov 13 03:48 x.c.000
-rw-rw-r-- 1 whm whm 50 Nov 13 03:48 x.c.001
-rw-rw-r-- 1 whm whm 38 Nov 13 03:48 x.c.002

% cat x.c.*
#include <stdio.h>

int main()
{
   char line[100];

   while (fgets(line, sizeof(line), stdin) != NULL)
      fputs(line, stdout);
}
```

split0.c, continued

```
int main(int argc, char *args[])
{
    FILE *in = fopen(args[1], "r");
    int size = atoi(args[2]), n = 0;

    while (1) {
        char buf[size];  // shortcut: size the buffer so that one fread does it.
        int bytes = fread(buf, sizeof(char), sizeof(buf), in);
        if (bytes == 0)
            break;

        char name[strlen(args[2])+20];
        sprintf(name, "%s.%03d", args[1], n++);
        FILE *out = fopen(name, "w");

        fwrite(buf, sizeof(char), bytes, out);
        fclose(out);
        }
}
```

Sidebar: System calls

A *system call* is a direct interface to a service provided by the operating system kernel.

The library function **fopen** eventually uses the **open** system call to actually open a file. **fread** eventually uses the **read** system call actually read bytes.

System calls are described in section 2 of the manual, but library functions are described in section 3. We might say that **fopen(3)** uses **open(2)** to make it clear that **fopen** is a library function but **open** is a system call.

System calls are typically things that require special privileges. For example, **open(2)** takes file permissions into account. **kill(2)** takes process ownership into account before allowing a process to be killed.

System calls almost always have wrapper functions, so you can use a system call like any other function but the code resides in the kernel, not the library.

Do **man 2 intro** and **man 2 syscalls** to read more about system calls.

stat(2)

`stat` is a system call that returns information about a file. Here's the prototype for the wrapper function:

```
int stat(const char *path, struct stat *buf);
```

Here's the `stat` structure:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

stat(2), continued

Let's use **stat** to write a program that simply prints the sizes of files named on the command line:

```
% stat1 stat1.c stat1 /tmp/20m
stat1.c: 348 bytes
stat1: 10632 bytes
/tmp/20m: 168888897 bytes
```

Code: (Note the includes needed, as shown by **man 2 stat**.)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *args[]) // stat1.c
{
   struct stat statbuf;
   for (char **ap = args+1; ap < args+argc; ap++) {
      if (stat(*ap, &statbuf) == -1)
         perror(*ap);
      else
         printf("%s: %zu bytes\n", *ap, statbuf.st_size);
      }
}
```

stat(2), continued

stat2.c uses the stat-specific macros **S_ISREG** and **S_ISDIR** to classify files as regular files, directories, or "other".

```
int main(int argc, char *args[])
{
    struct stat statbuf;
    for (char **ap = args+1; ap < args+argc; ap++) {
        if (stat(*ap, &statbuf) == -1)
            perror(*ap);
        else {
            char *type;
            if (S_ISREG(statbuf.st_mode))
                type = "regular file";
            else if (S_ISDIR(statbuf.st_mode))
                type = "directory";
            else
                type = "other";
            printf("%s: %zu bytes (%s)\n",
                *ap, statbuf.st_size, type);
            }
        }
}
```

Usage:

```
% stat2 stat1.c ..  /dev/null
stat1.c: 348 bytes (regular file)
..: 110 bytes (directory)
/dev/null: 0 bytes (other)
```

Stream positioning

Input and output streams have a notion of current position. **ftell(3)** returns the *file position indicator* for a stream. Here's its prototype:

```
long ftell(FILE *stream);
```

Here's a program that queries the position of a stream after reading each line:

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

int main(int argc, char **argv) // ftell1.c
{
   FILE *f = fopen(argv[1], "r");
   char *result = 0;
   ssize_t bytes_read; size_t bytes_allocd;

   while ((bytes_read = getline(&result, &bytes_allocd, f))
         != EOF) {
      fprintf(stdout, "pos: %ld\n", ftell(f));
      }
}
```

Usage:

```
% ftell1 ftell1.c
pos: 20
pos: 39
pos: 40
pos: 72
pos: 74
...
```

Stream positioning, continued

fseek(3) can be used to set the file position indicator. Here's the prototype:

```
int fseek(FILE *stream, long offset, int whence);
```

Here's a program that "seeks" to a specified position in a file and then reads and displays some number of bytes beginning at that position:

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

int main(int argc, char *argv[])
{
    FILE *f = fopen(argv[1], "r");
    long offset = atoi(argv[2]);
    int length = atoi(argv[3]);
    fseek(f, offset, SEEK_SET);

    char buf[length];
    fread(buf, sizeof(char), length, f);
    fwrite(buf, sizeof(char), length, stdout);
    fputc('\n', stdout);
}
```

Usage:

```
% fseek1 fseek1.c 45 25
ain(int argc, char *argv[

% fseek1 $a6/a6/256.chars 97 10
abcdefghij

% fseek1 /tmp/20m 168888888 8
20000000
```

Stream positioning, continued

fseek1.c used a whence of SEEK_SET to go to an absolute position. The program below uses SEEK_CUR to do relative repositioning.

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

int main(int argc, char *argv[])
{
    FILE *f = fopen(argv[1], "r");
    long skiplen = atoi(argv[2]);

    int N = 3;
    while (1) {
        char buf[N];
        if (fread(buf, sizeof(char), N, f) != N)
            break;
        fwrite(buf, sizeof(char), N, stdout);
        fseek(f, skiplen, SEEK_CUR);
        fprintf(stdout, "...");
        }
    fprintf(stdout, "\n");
}
```

Usage: (output wraps)

```
% fseek2 fseek2.c 10
#in…io….e <…
in… ar…arg…
FI…pen…"r"…g s…toi…

 … 3;… (1…  c…;
 …
(fr…ize…N, …   …eak…fwr…iz
e…N, …   …f, …EEK…   …std…
);
…   …tdo…
}
…
```

Stream positioning, continued

fseek can be used when writing to files, too. The following program writes the digits 0-9, skiping 999,999 bytes between each digit.

```
int main(int argc, char *argv[])
{
    FILE *f = fopen(argv[1], "w");
    for (int i = 0; i < 10; i++) {
        fprintf(f, "%d", i);
        fseek(f, 999999, SEEK_CUR);
        }
    fclose(f);
}
```

Usage:

```
% fseek3 big
% ls -l big
-rw-rw-r-- 1 whm whm 9000001 Nov 15 18:32 big
% fseek1 big 0 1
0
% fseek1 big 1000000 1
1
% fseek1 big 9000000 1
9
```

Stream positioning, continued

At hand:

```
% fseek3 big
% ls -l big
-rw-rw-r-- 1 whm whm 9000001 Nov 15 18:32 big
% fseek1 big 5000000 1
5
```

Speculate: What data is between those digits?

```
% fseek1 big 4999998 5 | cat -A
^@^@5^@^@$
% fseek1 big 4999998 5 | $fall15/a6/vis
<#0><#0>5<#0><#0><NL>
```

If we seek beyond the end of a file and then write data, UNIX fills the gap with zero-valued bytes.

Let's check that by using **tr** to delete **NUL**s and see what's left:

```
% tr -d \\0 < big
0123456789%
```

Stream positioning continued

For reference:

```
% fseek3 big
% ls -l big
-rw-rw-r-- 1 whm whm 9000001 Nov 15 18:45 big
```

du(1) shows how much disk space is being used by a file. The **-h** flag says to report in human-readable form. See anything odd?

```
% du -h big
1.3M  big
```

Zero-filled sections created by seeks and writes past the end of a file aren't necessarily "there"! It can be said that **big** is a *sparse file*.

Here's something odd:

```
% cp big big2
% du -h big*
1.3M  big
512   big2
% cp big2 big3
% du -h big*
1.3M  big
8.7M  big2
512   big3
```

A few minutes later...

```
% du -h big*
1.3M  big
8.7M  big2
8.7M  big3
```

The **popen** function

A very useful, but non-C11-standard stream I/O function is **popen**:

```
FILE *popen(const char *command, const char *type);
```

command is a shell command line. The shell is invoked to run the line.

If **type** is **"r"** then reading from the stream produces lines from the standard output stream of the command. If **"w"**, then writes to the stream become the standard input of the command.

This program uses **popen** to count users by reading the output of **who:**

```
int main() { // popen1.c
   FILE *who = popen("/usr/bin/who", "r");
   int c, users = 0;

   while ((c = fgetc(who)) != EOF)
      if (c == '\n') users++;

   pclose(who);
   fprintf(stdout, "%d users currently logged in\n", users);
}
```

popen, continued

Let's use **popen** to implement a simplified version of **picklines**: no negative values.

The **ed** editor's **p** command prints lines. Examples: **7p**, **3,20p** Let's use **popen** to send **p** commands to **ed**!

```
int main(int argc, char **argv)
{
    char cmd[strlen(argv[1])+15];
    sprintf(cmd, "/bin/ed -s %s", argv[1]);
    FILE *ed = popen(cmd, "w");

    argv += 2;
    while (*argv) {
        char *p = strchr(*argv, ':');
        if (p) *p = ',';
        fprintf(ed, "%sp\n", *argv++);
        }

    pclose(ed);
}
```

Usage:

```
% plines popen1.c 1:3 6 9
#include <stdio.h>

int main()
    int c, users = 0;
        if (c == '\n')
```

The **system** function

popen lets us communicate with a command but sometimes we just want to run a command. The **system** function is good for that. Here's the prototype:

```
int system(const char *command);
```

Imagine a program that creates a tree of temporary files in the current directory. When it's done, it could call **system** to remove that directory:

```
int main(int argc, char *args[])
{
    ...lots...
    if (system("rm -r .program_tmp 2>/dev/null") != 0)
        fprintf(stderr, "Warning: cleanup failed\n");
}
```

We use redirection to discard error output from **rm**. We check **rm**'s exit code, returned by **system**, to see if the cleanup failed for some reason.

Do we need both **popen** and **system**?

- Could we implement the above using **popen** instead of **system**?
- Could we implement our **plines.c** using **system** instead of **popen**?

The **strace** command

The **strace** command can be used to observe the system calls made by a program. Here's a simple test:

```
int main() // strace2.c
{
   printf("IN MAIN\n");  // so we can see when we get to here
   char *p = malloc(1);
}
```

Let's run it with strace:

```
% strace strace2
execve("/home/whm/cw/c/strace2", ["strace2"], [/* 156 vars */]) = 0
...lots...
write(1, "IN MAIN\n", 8IN MAIN
)            = 8
brk(0)                          = 0x1bf9000
brk(0x1c1a000)                  = 0x1c1a000
exit_group(0)                   = ?
```

Note that **printf** ends up invoking the **write** system call. The calls to **brk** are caused by **malloc**—it's apparently expanding the data segment size by 132K.

strace, continued

```
int main(int argc, char *args[])
{
   printf("IN MAIN\n");
   FILE *f = fopen(args[1], "r+");
   printf("FOPEN DONE\n");
   char c = fgetc(f);
   fseek(f, -1, SEEK_CUR);
   fputc(c, f);
   fclose(f);
}
```

CORRECTED

```
write(1, "IN MAIN\n", 8IN MAIN
)             = 8
brk(0)                               = 0x24f8000
brk(0x2519000)                       = 0x2519000
open("x.c", O_RDWR)                  = 3
write(1, "FOPEN DONE\n", 11FOPEN DONE
)          = 11
fstat(3, {st_mode=S_IFREG|0644, st_size=1048, ...}) = 0
mmap(NULL, 1048576, PROT_READ|PROT_WRITE,
 MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc617e6d000
read(3, "#define _GNU_SOURCE\n#includ"..., 1048576) = 1048
lseek(3, -1048, SEEK_CUR)            = 0
write(3, "#", 1)                     = 1
close(3)                             = 0
munmap(0x7fc617e6d000, 1048576)      = 0
exit_group(0)                        = ?
```

I/O buffering

Here's **buffer1.c**:

```
int main(int argc, char *args[])
{
   printf("IN MAIN\n");
   int c;
   while ((c = getchar()) != EOF)
      putchar(c);
}
```

Execution with **strace**:

```
% seq 10 | strace buffer1 > x
...lots...
read(0, "1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n", 4096) = 21
read(0, "", 4096)                        = 0
write(1, "IN MAIN\n1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n", 29) = 29
```

What's surprising about the trace?

The code first does a **printf** and then alternately reads and writes characters, but the trace shows one big read and one big write!

This is I/O *buffering* in action!

I/O buffering, continued

For reference:

```
% seq 10 | strace buffer1 > x
...lots...
read(0, "1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n", 4096) = 21
read(0, "", 4096)                         = 0
write(1, "IN MAIN\n1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n", 29) = 29
```

The motivation for buffering is that system calls like **read** and **write** are relatively slow and resource intensive—there's a *context switch* from *user mode* to *kernel mode* and back.

With buffering we batch many program-level input or output operations into a single system-level input or output operation, sometimes reducing the number of system calls by factors of hundreds or thousands.

I/O buffering, continued

buffer2.c uses the **read** and **write** system calls directly. File descriptors 0 and 1 are standard input and standard output, respectively.

```
#include <unistd.h>
int main()
{
    char c;
    while (read(0, &c, 1) != 0)
        write(1, &c, 1);
}
```

buffer1.c, for reference:

```
int main(int argc, char *args[])
{
    printf("IN MAIN\n");
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

Let's compare the two:

```
% seq 1000000 > 1m

% time buffer1 < 1m > x
real   0m0.446s
user   0m0.184s
sys    0m0.012s

% time buffer2 < 1m > x
real   0m6.176s
user   0m0.384s
sys    0m5.580s
```

Where's the big difference?

In "system" time—time spent in the kernel on the program's behalf.

I/O buffering, continued

For reference:

```
% seq 10 | strace buffer1 > x
...lots...
read(0, "1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n", 4096) = 21
read(0, "", 4096)                         = 0
write(1, "IN MAIN\n1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n", 29) = 29
```

Is the user aware of buffering?

The user expects to see "IN MAIN" followed by 1 to 10 end up in **x**, and that's exactly what they see.

I/O buffering, continued

Instead of writing to a file with **seq 10 | strace buffer1 > x**, let's write to the screen. Program output is shown in bold and underlined.

```
% echo $'just\ntesting\nthis' | strace buffer1
write(1, "IN MAIN\n", 8IN MAIN
)            = 8
read(0, "just\ntesting\nthis\n", 4096)  = 18
write(1, "just\n", 5just
)              = 5
write(1, "testing\n", 8testing
)            = 8
write(1, "this\n", 5this
)              = 5
read(0, "", 4096)                    = 0
exit_group(0)                        = ?
```

What's different?

A **write** is done for each line. That's because when standard output is the screen, output is *line buffered* by default, so that the user will see a line at a time.

Exercise:

Try **strace buffer1** (no piping or redirection) and see how things differ.

I/O buffering, continued

Here's a program that prints a N-second countdown and then "Go!":

```
int main(int argc, char **argv) // buffer3.c
{
    for (int i = atoi(argv[1]); i >= 1; i--) {
        fprintf(stdout, "%d...", i);
        sleep(1);
        }

    printf("Go!\n");
}
```

Let's run it live.

```
% buffer3 5
5...4...3...2...1...Go!
```

Because standard output is line-buffered, the user doesn't see any output until a newline is output.

I/O buffering, continued

We can force the countdown to appear second by second by *flushing* the stream.

```
int main(int argc, char **argv) // buffer3a.c
{
   for (int i = atoi(argv[1]); i >= 1; i--) {
      fprintf(stdout, "%d...", i);
      fflush(stdout);        // added
      sleep(1);
      }

   printf("Go!\n");
}
```

When a stream is associated with the screen, **fflush** causes output to immediately appear.

When a stream is associated with a file, ensuring that data has been physically written to the disk is more involved. The details begin with **fsync(2)**.

Start with **man setbuf** to learn more about buffering.

Some things I should have talked about sooner...

unsigned values

typedef

const

Unsigned types

Each of the integral types in C has an *unsigned* counterpart that has the same size but no sign. An unsigned instance of a type is declared by adding **unsigned** to the type.

On lectura, a two-byte, 16-bit signed **short** has the range -32,768 to 32767. (**SHRT_MIN** to **SHRT_MAX**.)

An unsigned **short** is declared like this:

```
unsigned short i;
```

Or this:

```
short unsigned i;
```

The variable **i** declared above has the range 0 to 65535 (**USHRT_MAX**).

C11 requires this:

sizeof(unsigned *TYPE*) == sizeof(*TYPE*)

Unsigned types, continued

With an unsigned type, arithmetic is odometer-like: Adding one to the largest number produces zero. Subtracting one from zero yields the largest number.

Example:

```
unsigned short i = USHRT_MAX;  // 65535

printf("i = %u\n", i);
   // Output: i = 65535

i++;
printf("i = %u\n", i);
   // Output: i = 0

i--;
printf("i = %u\n", i);
   // Output: i = 65535

i = 32771;  // 32768 + 3 == 32771
i *= 4;
printf("i = %u\n", i);
   // Output: i = 12
```

Unsigned types, continued

Assignment of an unsigned value to a signed variable produces the same value if the unsigned value can be represented by the variable.

```
unsigned int ui = 1000000000;  // one billion
int si = ui;

printf("si = %d\n", si);
     // Output: si = 1000000000
```

If the unsigned value can't be represented by the signed variable the end result is a "wrapped around" negative value in the signed variable:

```
ui = 3000000000U;  // Trailing 'U' avoids a warning
si = ui;
printf("si = %d\n", si);
     // Output: si = -1294967296
```

However, both **si** and **ui** have the same internal representation:

```
printf("si = %x, ui = %x\n", si, ui);
     // Output: si = b2d05e00, ui = b2d05e00
```

Unsigned types, continued

Another case:

```
int si = -1;

unsigned int ui = si;

printf("ui = %u\n", ui);
    // Output: ui = 4294967295 (max value)

printf("si = %x, ui = %x\n", si, ui);
    // Output: si = ffffffff, ui = ffffffff

si++;
ui++;
printf("si = %d, ui = %u\n", si, ui);
    // Output: si = 0, ui = 0
```

Unsigned types, continued

The basic concept of an unsigned quantity is simple but some complex and sometimes counterintuitive rules come into play when signed and unsigned quantities are mixed.

Example:

```
int i = -1;
unsigned int j = 1;
printf("i < j = %d\n", i < j);
   // Output: i < j = 0
```

i (-1) is not less than j (1)!

Why: i is converted to an unsigned int (4294967295) yielding

```
        i              <         j
   4294967295                    1
```

A simple rule of thumb for novices:

Use unsigned types only when you must.

Unsigned types, continued

Again, a simple rule of thumb for novices:

Use unsigned types only when you must.

One situation in which you must use an unsigned type is when the language or a library routine dictates it. Two examples:

C11 specifies that **sizeof** produces an unsigned integer value with implementation-specific size. With **gcc** on lectura it is an **unsigned int**.

The sizing parameters of **malloc**, **calloc**, and **realloc** are unsigned integers.

Another situation where an unsigned type is appropriate is when a low-level representation of data is unsigned. For example, if a network packet has a 16-bit length field, then representing that field with an **unsigned short** is a good choice, assuming that a **short** is 16 bits.

char can be signed or unsigned

C11 says that whether **char** is signed or unsigned is implementation-specific!

Let's see what we've got:

```
iprint((char)0 < (char)150);
iprint((unsigned char)0 < (unsigned char)150);
iprint((signed char)0 < (signed char)150);
```

Output:

```
(char)0 < (char)150 is 0
(unsigned char)0 < (unsigned char)150 is 1
(signed char)0 < (signed char)150 is 0
```

Here's another way to confirm that with **gcc** on lectura, **char** is **signed char**.

```
iprint(CHAR_MIN);
iprint(CHAR_MAX);
```

Output:

```
CHAR_MIN is -128
CHAR_MAX is 127
```

char can be signed or unsigned, continued

The following program attempts to tally byte values in the first 4K of **stdin**. What's wrong with it?

```
int main()  // uns2.c
{
   int counts[256];
   char buf[4096];
   fread(buf, sizeof(char), sizeof(buf), stdin);

   for (char *p = buf; p < buf + sizeof(buf); p++)
      counts[*p]++;
}
```

There's a warning on **counts[*p]++**:

```
array subscript has type ‘char’ [-Wchar-subscripts]
```

Byte values above 127 will be treated as negative values. The counts will end up in **counts[-128]** through **counts[127]**!

typedef

The **typedef** *specifier* is used to create new datatype names.

For example,

```
typedef int integer;
```

makes **integer** a synonym for **int**.

This specification essentially extends the language by creating a new type, named **integer**, that can be used at any place in code where a type is expected.

Given the above **typedef**, the following code is valid:

```
integer i;
integer *a[10];
i = (integer)2.5;
integer *p = &i;
sizeof(integer)
```

Note that a **typedef** alters the syntax of C! The above code would be syntactically invalid if **typedef int integer;** didn't precede it.

typedef, continued

typedefs let us create type names that express qualities and/or typical usage of values of that type.

For example, the standard specifies that **sizeof** produces a value of type **size_t**. It requires that **size_t** be an unsigned integer type but it doesn't require it to be any particular size, like **int** or **long**.

The implementation, **gcc** on lectura for us, provides a **typedef** in **<stddef.h>**:

```
typedef long unsigned int size_t;
```

Functions that work with sizes of memory objects are then written using **size_t**:

```
size_t strlen(const char *s);
void *malloc(size_t size);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

typedefs let us rendezvous on a type name but avoid a possibly burdensome requirement about the range of values.

The standard says to use **size_t**. The implementation says what **size_t** is.

typedef, continued

There are lots of **typedef**s.

The difference of two pointers is a **ptrdiff_t**:

```
typedef long int ptrdiff_t;
```

<pwd.h> uses **uid_t**, which is defined with an intermediate **typedef**, **__uid_t**:

```
typedef  unsigned  int  __uid_t;
typedef  __uid_t  uid_t;
```

getline(3) returns an **ssize_t**, a signed quantity:

```
typedef  long  int  __ssize_t;
typedef  __ssize_t  ssize_t;
```

A convention with types used by library functions is that **typedef**'d names end with **_t** but the standard does not require that.

typedef, continued

Microsoft's Win32 API relies heavily on **typedef**s. Here are a few of them:

```
typedef char *PSZ;
typedef unsigned long DWORD;
typedef int BOOL;
typedef unsigned char BYTE;
typedef BYTE *PBYTE;
typedef unsigned short WORD;
typedef int *LPINT;
typedef unsigned int UINT;
typedef unsigned int *PUINT;
```

Given the above, what does **PBYTE x;** declare?

x is pointer to **unsigned char.**

Function prototypes in the Win32 API use the defined types. One example:

```
HICON CreateIconFromResource(PBYTE presbits,
                DWORD dwResSize, BOOL fIcon, DWORD dwVer);
```

typedef and structures

Recall that the following declaration defines a structure tag named **Rectangle**.

```
struct Rectangle {
   int width;
   int height;
   };
```

To define an instance of the structure, both **struct** and the tag name must be used:

```
struct Rectangle r1;
```

A **typedef** can be used to create a **Rectangle** type based on the structure tag:

```
typedef struct Rectangle Rectangle;
```

With the **typedef** in place, **Rectangle** can be used like any other type:

```
Rectangle r, *rp, rects[5];

int area_rectangle(Rectangle *p);

Rectangle rotate_rectangle(Rectangle r);
```

typedef and structures, continued

Some variations are possible. One is to define a type, but not a tag:

```
typedef struct {
   double x,y;
   } Point;
```

Given the above, **Point** can be used as type, but not a tag:

```
Point p1;           // OK
struct Point p2;  // Error: storage size of 'p2' isn't known
```

This specification defines both a tag and a type:

```
typedef struct Point {
   double x,y;
   } Point;
```

Here's a self-referential structure. The tag is required.

```
typedef struct Node {
   int value;
   struct Node *next;
   } Node;
```

typedef and structures, continued

Slide 432 shows a simplified **typedef** for **FILE**:

```
typedef struct _IO_FILE {
    ...lots...
    } FILE;
```

Here's what **<stdio.h>** actually does:

```
struct _IO_FILE;

typedef struct _IO_FILE FILE;

struct _IO_FILE {
  int _flags;
  char* _IO_read_ptr;
  char* _IO_read_end;
  ...lots...
  };
```

typedef scope

A **typedef** has scope just like a variable:

```
typedef int GLOBAL_INT;    // Usable in all code that follows

void f()
{
   typedef int LOCAL_INT;   // Usable only in f()

   GLOBAL_INT i;
   LOCAL_INT j;
}

void g()
{
   GLOBAL_INT a;
   LOCAL_INT b;            // Error: LOCAL_INT is local to f()
}
```

The **const** qualifier

The **const** *qualifier* can be added to a declaration to indicate that a value is not to be changed.

The simplest use of **const** is to apply it to a scalar variable:

```
const int couple = 2;
```

An attempt to change **couple** produces a compilation error:

```
couple = 3;  // error: assignment of read-only variable 'couple'

couple++;   // error: increment of read-only variable 'couple'
```

Here's one way to view **const int couple = 2**:

"I do not intend to change **couple**. Stop me if I try to change it."

The Java counterpart for **const** is **final**:

```
final int couple = 2;
```

const, continued

const can be applied to the object referenced by a pointer:

```
char s[] = "abc";
const char *p = s;   // "Stop me if I try to change *p"
```

An expression that would change the object referenced by ***p** produces a compilation error:

```
*p = 'x';      // error: assignment of read-only location

p[0] = 'x';    // error: assignment of read-only location

(*p)++;        // error: increment of read-only location
```

Surprise: I can't use ***p** to make a change but I can change the contents of **s** directly!

```
s[0] = 'x';
*s = 'x';
```

How could the two changes above be prohibited?

```
const char s[] = "abc";
```

const, continued

At hand:

```
char s[] = "abc";
const char *p = s;    // "Stop me if I try to change *p"
```

Note that it is only ***p** that is considered to be read-only. The value of **p** can be changed:

```
p = "xyz";              // OK
char c = *p++;          // OK
```

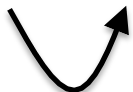
Speculate: how could we make **p** "unchangeable", too?

```
const char *const p = s;
```

To understand what **const** applies to, look at the next word:

```
const char *const p = s;
```

```
const int couple = 2;
```



const in parameters

Perhaps the most common use of **const** is to apply it to pointer parameters. Here's the prototype for **strlen**:

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

It promises that **strlen** will change no characters in the string whose length is being calculated.

Here is **strcpy**:

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

Which way does the data move in the following function?

```
void movebytes(const void *p1, void *p2, size_t n);
```

Because the first parameter is **const void *** and the second is **void ***, it appears that **p1** is the source and **p2** is the destination.

const in parameters, continued

const pointer parameters provide two benefits:

- The prototype informs a user whether the function might modify data referenced by a pointer.

- The author of the function is notified of a violation of that promise at compile time.

Here's a function with an error that is caught at compile-time:

```
void copy(char *s1, const char *s2)
{
    while (*s2++ = *s1++)
        ;
}
```

const in parameters, continued

Consider this string length function:

```
int length(const char *s)
{
   char *s0 = s;  // warning: initialization discards qualifiers
                  // from pointer target type

   while (*s++)
      ;

   return s - s0 - 1;
}
```

What's the problem?

- I first say that I don't intend to change characters referenced by **s**.
- I then declare **s0**, a copy of **s** with which I can change the referenced characters.
- The compiler warns me of the apparent contradiction.

const in parameters, continued

If **const** is used in some cases but not others, it can create headaches. Example:

```
int twice(int *p)
{
   return *p * 2;
}

int f(const int *p)
{
   return twice(p);   // warning: passing arg 1 of 'twice' discards
                      // qualifiers from pointer target type
}
```

As rule, developers collaborating on a body of code of need to decide as a group whether to use **const**.

Casting away **const**

A **const** qualification can be cast away. This code compiles and runs with **gcc** on lectura:

```
int main() // const3.c
{
    const int four = 4;

    int *p = (int*)&four;

    *p = 10;

    printf("four = %d\n", four);
        // Output: four = 10
}
```

However, the standard says this:

> "If an attempt is made to modify an object defined with a **const**-qualified type through use of an L-value with non-**const**-qualified type, the behavior is undefined."

Enumerations

An **enum** declaration defines a set of named integer constants. Example:

```
typedef enum { Open, Closed, Unknown } ValvePosition;
```

By default, a value of zero is assigned to the first name. Following names are given the value of the preceding name plus one. In this case, **Open** is 0, **Closed** is 1, **Unknown** is 2.

Usage:

```
ValvePosition vpos = Closed;
printf("vpos = %d\n", vpos);  // Output: vpos = 1
```

Another example:

```
ValvePosition vpos = sense_valve(valve_num);

switch (vpos) {
   case Open: ...
   case Closed: ...
   case Unknown: ...
}
```

Enumerations, continued

A common use of enumerations is to produce a sequence of integer constants. Numbering in the following **enum** starts at 1, and then has skips to 125 and 140.

```
enum AtomicNumbers { Hydrogen=1, Helium, Beryllium, Boron, Carbon,
    ...lots more...
    Unobtainium = 125, Eludium, Hardtofindium = 140, Impossibrium,
    Buzzwordium, Phlebotinum };
```

With **#define**, we'd have to provide each number, which could be error-prone.

```
#define Hydrogen     1
#define Helium       2
#define Beryllium    3
...
```

The names in an **enum** must be unique, but the values do not.

enums are somewhat between **#define**s and **const**s: the values are considered to be constants (and can be used in a **switch**, for example) but **enum**s are scoped.

Unions

Unions

A *union* can be thought of as a structure in which all members start at the same address.

A **union** declaration looks like a structure declaration but with the **union** keyword instead of "**struct**":

```
typedef union {
    int i;
    long L;
    unsigned char c;
    unsigned char bytes[sizeof(long)];
    } type_u;

type_u U;
```

Another way to think about it:
U might be an **int i**, or a **long L**, or an **unsigned char c**, or ...

The size of a **union** is the size of its largest member.

What is **sizeof(U)**?

8

Which members determine the size of an instance of **type_u**?

L and **bytes** are both eight bytes in size.

Unions, continued

At hand:

```
typedef union {  // union1.c
   int i;
   long L;
   unsigned char c;
   unsigned char bytes[sizeof(long)];
   } type_u;

type_u U;  // sizeof(U) is 8
```

Problem: Prove that all members start at the same address.

Let's use **paddr** from **352.h**:

```
paddr(&U.i);
paddr(&U.L);
paddr(&U.c);
paddr(&U.bytes);
paddr(&U);
```

Output:

```
&U.i = 140732672934400
&U.L = 140732672934400
&U.c = 140732672934400
&U.bytes = 140732672934400
&U = 140732672934400
```

Union example: robot control

A common technique is to have a structure that consists of a type and a union. The type indicates how to interpret the union.

Imagine a simple robot with three commands: move, rotate camera, and beep.

Here is a structure to represent commands. It has two fields: **type** and **command**.

```
typedef struct {
   enum CmdType { Move, RotateCam, Beep } type;

   union {
      struct MoveCommand {
         char direction; // N, S, E, W
         int distance;
         } move;

      struct RotateCamCommand {
         int degrees;
         } rotate_cam;

      struct BeepCommand {
         char sequence[MAX_BEEP+1];
         } beep;
      } command;

   } RobotCommand;
```

Robot control, continued

Let's make three commands and send them to the robot.

```
RobotCommand cmds[3], *cmdp = cmds;

cmdp->type = Move;
cmdp->command.move.direction = 'N';
cmdp->command.move.distance = 10;
cmdp++;

cmdp->type = RotateCam;
cmdp->command.rotate_cam.degrees = 180;
cmdp++;

cmdp->type = Beep;
strcpy(cmdp->command.beep.sequence,
    "321__248");
cmdp++;

send_commands(cmds, 3);   // send via communication link
```

```
typedef struct {
    enum CmdType { ... } type;
    union {
        struct MoveCommand {
            char direction;
            int distance;
            } move;
        struct RotateCamCommand {
            int degrees;
            } rotate_cam;
        struct BeepCommand {
            char sequence[MB+1];
            } beep;
        } command;
    } RobotCommand;
```

Using a union reduces the amount of data that needs to be transmitted.

Robot control, continued

The robot processes each command with this routine:

```
void process_command(RobotCommand *cmd)
{
   switch (cmd->type) {
    case Move:
     do_move(cmd->command.move.direction,
         cmd->command.move.distance);
     break;
    case RotateCam:
     do_rotate(
          cmd->command.rotate_cam.degrees);
     break;
    case Beep:
     do_beep(cmd->command.beep.sequence);
     break;
     }
}
```

```
typedef struct {
   enum CmdType { ... } type;
   union {
      struct MoveCommand {
         char direction;
         int distance;
         } move;
      struct RotateCamCommand {
         int degrees;
         } rotate_cam;
      struct BeepCommand {
         char sequence[MB+1];
         } beep;
      } command;
   } RobotCommand;
```

Architectural experiment

At hand:

```
typedef union {
    int i;
    long L;
    unsigned char c;
    unsigned char bytes[sizeof(long)];
    } type_u;
```

Experiment: (union2.c)

```
type_u U;

U.L = 0x1122334455667788;

printf("U.bytes[0] = %x\n", U.bytes[0]);
printf("U.bytes[7] = %x\n", U.bytes[7]);
```

Output:

```
U.bytes[0] = 88
U.bytes[7] = 11
```

The output demonstrates that the x86 is a *little-endian* architecture.

See Endianess on Wikipedia for more.

Unions in general

A common use-case for a union is to represent an array that's a heterogeneous collection of values.

With the robot example, the heterogeneous collection consisted of move, rotate camera, and beep commands.

How would we likely have implemented the robot example in Java?

With inheritance:

```
abstract class RobotCommand { ... }
class MoveCommand extends RobotCommand { ... }
class RotateCamCommand extends RobotCommand { ... }
class BeepCommand extends RobotCommand { ... }
```

Here's a union from the bash 4.2 source:

```
typedef union {
    int dest;               /* Place to redirect REDIRECTOR to, or ... */
    WORD_DESC *filename;    /* filename to redirect to. */
} REDIRECTEE;
```

Pointers to functions

Pointers to functions

It's possible to have a pointer to a function.

Here's a declaration for a variable named **fp**. The variable **fp** can hold a pointer to any function that takes two **int** arguments and returns an **int**:

```
int (*fp)(int, int);
```

Given this function,

```
int add(int a, int b)
{
   return a + b;
}
```

one can say:

```
fp = add;
int i = fp(5, 10);  // produces 15
```

The call **fp(5,10)** says "Call the function whose address is held in **fp** and pass it the arguments **5** and **10**."

Pointers to functions, continued

The declaration at hand:

```
int (*fp)(int, int);
    // fp points to a function that takes two ints and returns an int
```

The type of a function pointer includes both the type of the arguments and the return type of the function.

What's the error in each of the following?

```
int i = fp(7);
    // too few arguments to function

int i = fp(2, "3");
    // bad type for second argument

char *p = fp(3,4);
    // assigning int result to a pointer
```

Pointers to functions, continued

The declaration at hand:

```
int (*fp)(int, int);
    // fp points to a function that takes two ints and returns an int
```

Is the following assignment valid?

```
fp = strlen;
    // warning: assignment from incompatible pointer type
```

What's the type of **strlen**?

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

Problem: Write a declaration for **fp2** such that **fp2 = strlen** works.

```
size_t (*fp2)(const char *);
fp2 = strlen;
```

Is the following valid?

```
size_t (*fp3)(char *) = strlen;
    // warning: initialization from incompatible pointer type
```

Pointers to functions, continued

Here's an older, but still-valid form that you may encounter and/or prefer:

```
int (*fp)(int, int);

fp = &add;

int i = (*fp)(5, 10);
```

Note &add and indirection with (*fp).

Function pointers in structures

```
int add(int a, int b) { return a + b; }
int mult(int a, int b) { return a * b; }
// sub and div are similar

typedef struct {
   char op;
   int (*f)(int, int);
   } Op;

Op Ops[] = {
   { '+', add }, { '-', sub },
   { '*', mult }, { '/', div }, {} };

int main() // fp2.c
{
   int i, j; char op;
   while (scanf("%d %c %d", &i, &op, &j) == 3) {
      Op *p;
      for (p = Ops;  p->op;  p++)
         if (p->op == op) {
            int result = p->f(i, j);
            printf("%d\n", result);
            break;
            }
      }
}
```

Here's a calculator that accepts input like '3+5' and '71*82'. It looks up a function that corresponds to the operator and then calls the function via a function pointer.

Usage:

```
% a.out
 71 *  82
5822
```

What's needed to add an operation to the calculator?

Just add a function and an entry in **Ops**.

Is this code cleaner than it would be without using function pointers?

Functions that return function pointers

Here's a **typedef** for a function pointer that can reference the **add**, **sub**, etc. functions from the previous example:

```
typedef int (*binop)(int,int);  // Declares a type named 'binop'
```

Here's a function that looks up the function associated with a specified operator:

```
binop lookup(char op)
{
    for (Op *p = Ops; p->op; p++)
        if (p->op == op)
            return p->f;

    return 0;
}
```

For reference:

```
typedef struct {
    char op;
    int (*f)(int, int);
    } Op;

Op Ops[] = {
    { '+', add }, { '-', sub },
    { '*', mult }, { '/', div }, {} };
```

Standalone usage:

```
binop f = lookup('+');
int result = f(3,4);
```

Functions that return function pointers, continued

For reference:

```
typedef int (*binop)(int,int);
binop lookup(char op)
{
   for (Op *p = Ops; p->op; p++)
      if (p->op == op)
         return p->f;
   return 0;
}
```

Here's a revised **main** for the calculator:

```
int main()
{
   int i, j; char op;
   while (scanf("%d %c %d", &i, &op, &j) == 3) {
      binop opfcn = lookup(op);
      if (opfcn)
         printf("%d\n", opfcn(i, j));
      else
         puts("?");
      }
}
```

Sidebar: the **typedef** is a must!

Let's try it without the **typedef**:

```
% cat fp4a.c
int (*)(int,int) lookup(char op)
{
    for (Op *p = Ops; p->op; p++)
        if (p->op == op)
            return p->f;

    return 0;
}

% gcc fp4a.c
fp4a.c:1:7: error: expected identifier or '(' before ')' token
```

Fact: We can't write **lookup** without a **typedef**!

qsort

A number of library routines use function pointers. One of them is **qsort**, a sorting routine. Here's how the C11 standard describes it:

Synopsis

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Description

The **qsort** function sorts an array of **nmemb** objects, the initial element of which is pointed to by **base**. The size of each object is specified by **size**.

The contents of the array are sorted into ascending order according to a comparison function pointed to by **compar**, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

Once upon a time, I believe, **qsort** used the quicksort algorithm but the standard does not require that.

qsort, continued

Here's a program that sorts the characters in a string:

```
int main() // qsort2.c
{
    char s[] = "tim korb";
    qsort(s, strlen(s), sizeof(char), compare_chars);
    puts(s);    // Output:   bikmort
}
```

Problem: Write **compare_chars**.

```
int compare_chars(const void *vp1, const void *vp2)
{
    char c1 = *(char*)vp1;
    char c2 = *(char*)vp2;

    if (c1 < c2)
        return -1;
    else if (c1 == c2)
        return 0;
    else
        return 1;
}
```

Can we shorten **compare_chars**?

```
int compare_chars2(const void *vp1, const void *vp2)
{
    return *(char*)vp1 - *(char*)vp2;
}
```

qsort, continued

Here's a program that sorts lines read from standard input:

```
int main() { // qsort3.c
    char *lines[10000], **next = lines;
    ssize_t bytes_read; size_t bytes_allocd; char *result = 0;
    while ((bytes_read = getline(&result, &bytes_allocd, stdin)) != EOF) {
        *next++ = result;
        result = 0;
        }

    int nlines = next - lines;
    qsort(lines, nlines, sizeof(char*), compare_strs);

    for (next = lines; next < lines + nlines; next++)
        fputs(*next, stdout);
}
```

Problem: write **compare_strs**.

```
int compare_strs(const void *vp1, const void *vp2)
{
    const char *s1 = *(char**)vp1;
    const char *s2 = *(char**)vp2;

    return strcmp(s1, s2);
}
```

qsort, continued

Let's sort rectangles by decreasing area:

```
typedef struct { int w, h; } Rectangle;
int compare_rects(const void *vp1, const void *vp2)
{
   const Rectangle *rp1 = vp1; const Rectangle *rp2 = vp2;
   return rp2->w * rp2->h - rp1->w * rp1->h;
}
int main() { // qsort4.c
   Rectangle rs[] = {{3,4}, {5,1}, {10,2}, {3,3}};
   qsort(rs, sizeof(rs)/sizeof(rs[0]), sizeof(Rectangle), compare_rects);
}
```

Let's test with gdb:

```
(gdb) b 12
Breakpoint 1 at 0x400578: file qsort4.c, line 12.
(gdb) r
Breakpoint 1, main () at qsort4.c:12
12        qsort(rs, sizeof(rs)/sizeof(rs[0]), sizeof(Rectangle), compare_rects);
(gdb) n
13     }
(gdb) p rs
$1 = {{w = 10, h = 2}, {w = 3, h = 4}, {w = 3, h = 3}, {w = 5, h = 1}}
```

Arrays of multiple dimensions

Arrays of multiple dimensions—basics

C supports arrays of an arbitrary number of dimensions.

Here's a declaration for an array with two rows of three columns:

```
char a[2][3];
```

The elements of **a** are in consecutive memory locations. Arrays are stored in *row-major* form: The first row is the first three elements of the array; the second row is the second three elements. **sizeof(a)** is 6.

Here's code that prints the address of each element:

```
for (int r = 0; r < 2; r++) {
   for (int c = 0; c < 3; c++)
      printf("&a[%d][%d] = %lu  ", r, c, &a[r][c]);
   puts("");
   }
```

Output:

```
&a[0][0] = 6295608  &a[0][1] = 6295609  &a[0][2] = 6295610
&a[1][0] = 6295611  &a[1][1] = 6295612  &a[1][2] = 6295613
```

Sidebar: Pitfall with a[r,c]

Here's a pitfall:

```
for (int r = 0; r < 2; r++) {
    for (int c = 0; c < 3; c++)
        printf("&a[%d][%d] = %lu  ", r, c, &a[r,c]);  // should be &a[r][c]
    puts("");
    }
```

Output: (note the addresses are the same for both rows!)

```
&a[0][0] = 6295608  &a[0][1] = 6295611  &a[0][2] = 6295614
&a[1][0] = 6295608  &a[1][1] = 6295611  &a[1][2] = 6295614
```

In the expression **a[r,c]**, the construct **r,c** is a use of the comma operator, which evaluates both operands and produces the value of its right-hand operand.

a[r,c] does produce a warning:

```
left-hand operand of comma expression has no effect
```

At hand—an array with two rows of three columns:

```
char a[2][3];
```

View it this way:

```
a[0][0]   a[0][1]   a[0][2]
a[1][0]   a[1][1]   a[1][2]
```

Let's consider the types and sizes of `a`, `a[r]`, and `a[r][c]`:

The type of `a` is `char[2][3]`; `sizeof(a) == 6`.

What's the type and size of `a[r]`?

```
char[3]
sizeof(a[r]) == 3
```

What's the type and size of `a[r][c]`?

```
char
sizeof(a[r][c]) == 1
```

Another way to say it: `a` is an array of two arrays of three characters each.

Basics, continued

At hand—an array with two rows of three columns:

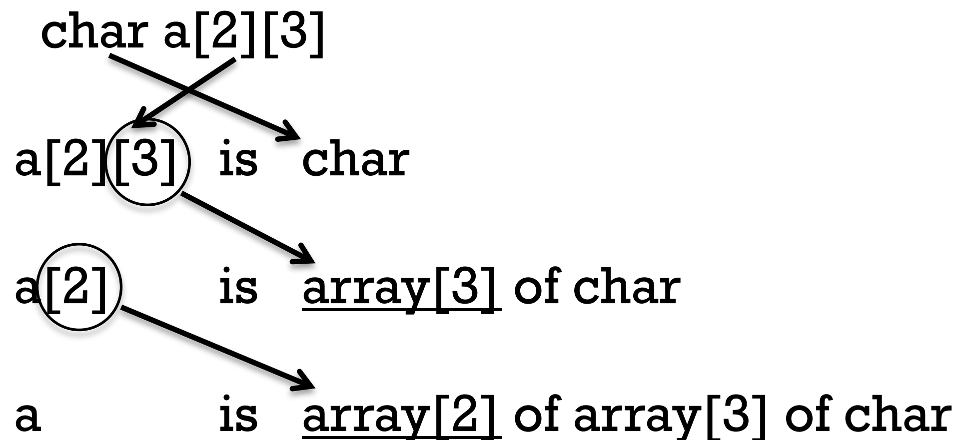
```
char a[2][3];
```

Note that the indexing operation is left-associative:

`a[r][c]` means `(a[r])[c]`

Think of `a[r][c]` as focusing first on a row and then on a column in that row.

Let's break down the type:



Basics, continued

At hand—an array with two rows of three columns:

```
char a[2][3];
```

Because **a[r]** is an array, **a[r]** is roughly equivalent to **&a[r][0]**, the address of the first byte of **a[r]**, the **r**'th row of **a**.

Therefore, **a[r]** can be used as a **char ***:

```
char *p = a[0];
strcpy(p, "a");
strcpy(a[1], "bc");
```

After the **strcpy** calls, **gdb** shows this:

```
(gdb) p a
$1 = {"a\000", "bc"}

(gdb) p/c a
$2 = {{97 'a', 0 '\0', 0 '\0'}, {98 'b', 99 'c', 0 '\0'}}
```

What's a way in which **a[r]** is not equivalent to **&a[r][0]**?

```
sizeof(a[r]) != sizeof(&a[r][0])
```

Basics, continued

The "outermost" dimension of an array can be established by the number of initializers present. Example: (**md4.c**)

```
char words[][15] = { "The", "C", "Programming", "Language", ""};
psize(words);
psize(words[0]);
puts("");

for (int w = 0;  w < sizeof(words) /sizeof(words[0] ); w++) {
   for (int pos = 0; pos < sizeof(words[0]); pos++) {
      char c = words[w][pos];
      putchar(c ? c : '.');
      }
   puts("");
   }
```

Output:

```
sizeof(words) is 75
sizeof(words[0]) is 15

The............
C..............
Programming....
Language.......
...............
```

Multiply-dimensioned parameters

A rule:

The outermost dimension of an array can be omitted in a function parameter.

Here's the printing loop of the previous example in the form of a function:

```
void print(char w[][15]) // md4a.c
{
    for (int r = 0; w[r][0] != '\0'; r++) {
        for (int c = 0; c < sizeof(w[0]); c++) {
            char ch = w[r][c];
            putchar(ch ? ch : '.');
            }
        puts("");
        }
}
```

Output:

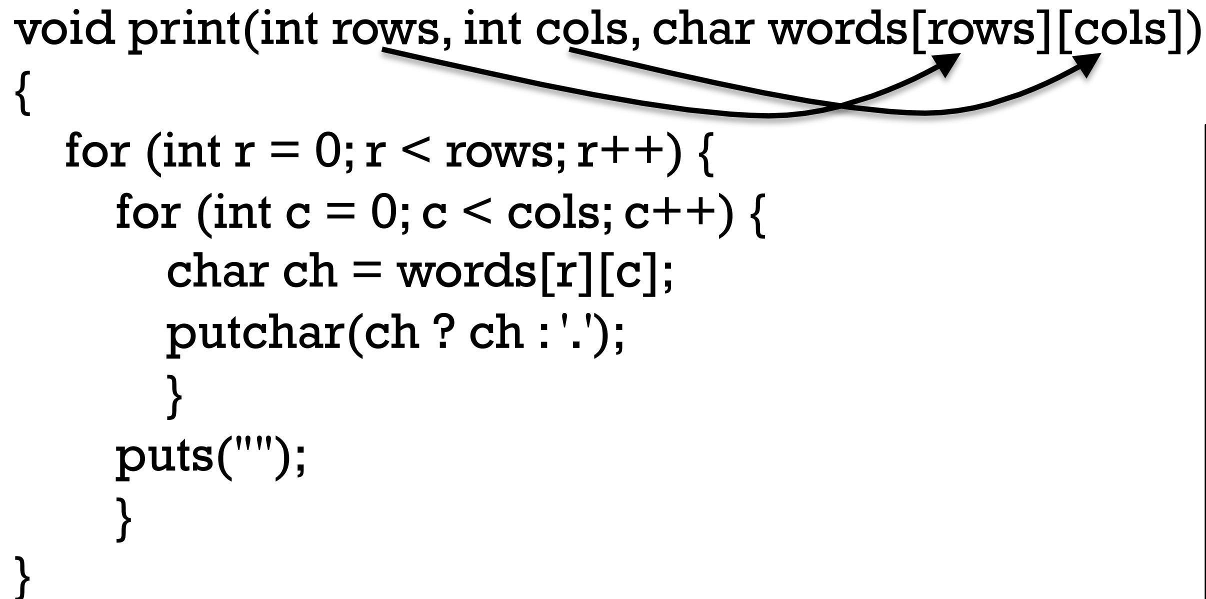
```
The............
C..............
Programming....
Language.......

Language.......
```

Usage:

```
char words[][15] = { "The", "C", "Programming", "Language", ""};
print(words);
puts("");
print(&words[3]);
```

Multiply-dimensioned parameters

The C11 standard allows an array parameter to be sized based on one or more preceding parameters. Here's a version of **print** that makes use of that.



```
void print(int rows, int cols, char words[rows][cols])
{
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            char ch = words[r][c];
            putchar(ch ? ch : '.');
            }
        puts("");
        }
}
```

Output:

```
The............
C..............
Programming....
Language.......

ab...
cde..
```

Usage: (note that the terminator, an empty string initializer, is no longer needed)

```
char w1[][15] = { "The", "C", "Programming", "Language"};

print(sizeof(w1)/sizeof(w1[0]), sizeof(w1[0]), w1);
puts("");

char w2[][5] = { "ab", "cde"};
print(sizeof(w2)/sizeof(w2[0]), sizeof(w2[0]), w2);
```

Higher dimensions

Here is a three dimensional array that represents a telephone keypad:

```
char phone_pad[4][3][7] = {
    {"1",      "2 ABC",  "3 DEF" },
    {"4 GHI",  "5 JKL",  "6 MNO" },
    {"7 PQRS", "8 TUV",  "9 WXYZ" },
    {"* TONE", "0 OPER", "#"}};
```

Note the types and sizes:

phone_pad	is	**char[4][3][7]**, size is 84
phone_pad[0]	is	**char[3][7]**, size is 21
phone_pad[0][0]	is	**char[7]**, size is 7
phone_pad[0][0][0]	is	**char**, size is 1

Higher dimensions, continued

For reference:

```
char phone_pad[4][3][7] = {
     {"1",      "2 ABC",  "3 DEF" },
     {"4 GHI",  "5 JKL",  "6 MNO" },
     {"7 PQRS", "8 TUV",  "9 WXYZ" },
     {"* TONE", "0 OPER", "#"}};
```

Some nested loops to print the contents:

```
for (int r = 0; r < sizeof(phone_pad)/sizeof(phone_pad[0]); r++) {
    for (int c = 0; c < sizeof(phone_pad[0])/sizeof(phone_pad[0][0]); c++) {
        printf("%-8s", phone_pad[r][c]);
        }
    puts("");
    }
```

Output:

```
1       2 ABC   3 DEF
4 GHI   5 JKL   6 MNO
7 PQRS  8 TUV   9 WXYZ
* TONE  0 OPER  #
```

Higher dimensions, continued

Here is an alternative formulation:

```
char *phone_pad[4][3] = {
    {"1", "2 ABC", "3 DEF" },
    {"4 GHI", "5 JKL", "6 MNO" },
    {"7 PQRS", "8 TUV", "9 WXYZ" },
    {"* TONE", "0 OPER", "#"}};
```

Let's interpret the type of **phone_pad**:

***phone_pad[4][3]**	**is**	**char**
phone_pad[4][3]	**is**	**pointer to char**
phone_pad[4]	**is**	**array [3] of pointer to char**
phone_pad	**is**	**array [4] of array [3] of pointer to char**

What is **sizeof(phone_pad)**?

96

And, that's not including space for the characters themselves!

What is **sizeof(phone_pad)** with the previous array? (**char phone_pad[4][3][7]**)

84

Sidebar: A notational alternative

Consider this program:

```
int main(int argc, char **argv)
{
   for (int i = 0; i < argc; i++)
      printf("%c\n", argv[i][i]);
}
```

Execution:

```
% a.out ab cde
a
b
e
```

How does it work?

x[y]	is equivalent to	*(x + y)
argv[i]	is equivalent to	*(argv + i)
argv[i][i]	is equivalent to	*(*(argv + i) + i)

Design considerations

When is it appropriate to use a multi-dimensional array?

When the data is "rectangular":

- Grids in games
- N-dimensional matrices
- And more...

As you'd expect, a multi-dimensional array can comprise values of any type.
Imagine a **volume** (three-dimensional) that comprises **Voxel** elements:

```
typedef struct {
    double density, temperature;
    time_t last_sample;
    Probe *probe;
    } Voxel;

Voxel volume[width][height][depth];
```

Multi-dimensional arrays are common in many domains but are relatively rare in systems programming.

More with complex declarations

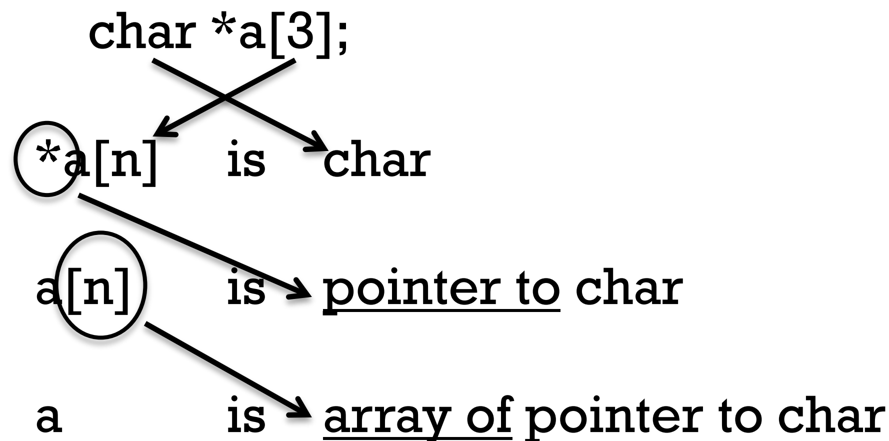
Review

We've learned this process for understanding declarations:

- Put the declaration expression on the left, the base type on the right, and "is" between them. (Remember: "declaration mimics use")

- Find the lowest precedence operator, remove it and prefix the right hand side with "**array[n] of**" or "**pointer to**" depending on whether the operator is [], *.

- Repeat until only the identifier remains.



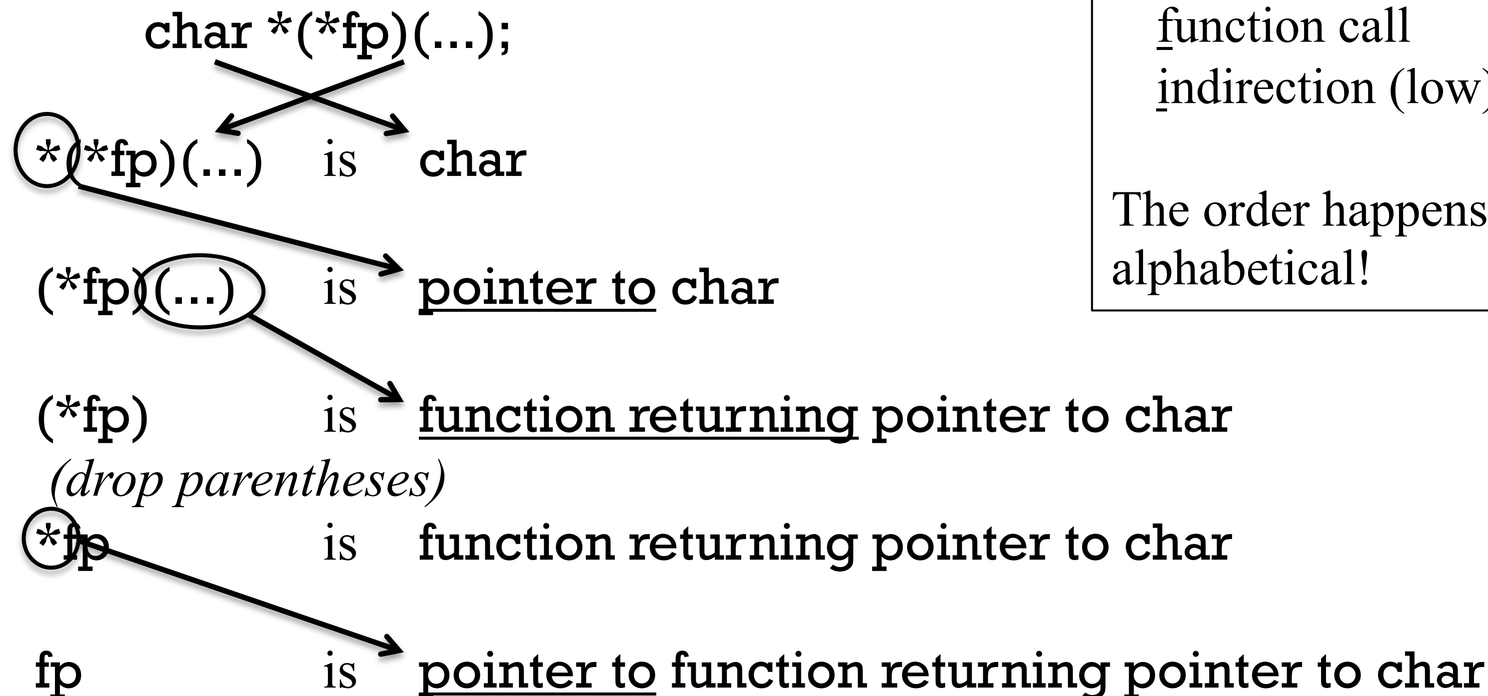
"function returning"

Let's add another rewriting rule: "function returning".

Example:

```
char *(*fp)(const char *, int c);
```

Parameter types are a separate question so we'll set the parameters aside.



Precedence:
- array subscripting (high)
- function call
- indirection (low)

The order happens to be alphabetical!

Example

A more interesting case:

```
char *(*sfuncs[5])(const char *, int);
```

Precedence:
- array subscripting (high)
- function call
- indirection (low)

Interpretation:

*(*sfuncs[5])(...)	is	char
(*sfuncs[5])(...)	is	pointer to char
*sfuncs[5]	is	function(...) returning pointer to char
sfuncs[5]	is	pointer to function(...) returning pointer to char
sfuncs	is	array [5] of pointer to function(const char *, int) returning pointer to char

Example, continued

At hand:

```
char *(*sfuncs[5])(const char *, int);
```

Full interpretation:

sfuncs is <u>array [5] of</u> pointer to function(const char *, int) returning pointer to char

Example of use:

```
char *(*sfuncs[])(const char *, int) = { strchr, strrchr};
int main()
{
    char *s = "banana";

    char *p1 = sfuncs[0](s, 'a');
    char *p2 = sfuncs[1](s, 'a');

    printf("%td %td\n", p1 - s, p2 - s);
}
```

Another example

Another:

```
char (*a[10])[5];
```

Precedence:
- <u>a</u>rray subscripting (high)
- <u>f</u>unction call
- <u>i</u>ndirection (low)

Interpretation:

(*a[10])[5]	is	char
(*a[10])	is	<u>array[5]</u> of char
*a[10]	is	array[5] of char
a[10]	is	<u>pointer to</u> array[5] of char
a	is	<u>array[10]</u> of ptr to array[5] of char

Note the sizes:

sizeof((*a[i])[j])	1
sizeof(*a[i])	5
sizeof(a[i])	8
sizeof(a)	80

English to C

The process can be reversed to go from English to C.

- Put the object name on the left and the stylized English description on the right.

- Remove the leftmost element of the English description and add the appropriate operator to the left hand side. Repeat until only the base type remains.

- If the operator being added is greater in precedence than the lowest precedence operator currently in the left hand side, first wrap the left hand side in parentheses.

Problem: Define p to be a pointer to an array of three characters.

```
p         is   pointer to array[3] of char
*p        is   array[3] of char
(*p)[3]   is   char
```

Declaration: `char (*p)[3];`

Note the third step: `*p` is wrapped in parentheses because the operator being added, `[3]`, is greater in precedence than the lowest precedence operator present (the indirection).

Given `p` as declared above, then if `p = 0`, what does `p++` do?

Example

Problem:

Declare an array `a` of five pointers to functions that take a `char` and returning an `int`.

Solution:

a	is	array[5] of pointer to function returning int
a[5]	is	pointer to function returning int
*a[5]	is	function returning int
(*a[5])(...)	is	int

(Note addition of parentheses.)

Declaration:

```
int (*a[5])(char);
```

cdecl.org

As mentioned on slide 326, some systems have **cdecl(1)**, but **cdecl** is web-enabled at **cdecl.org**.

Bit manipulation

Just a bit?

In some cases representing information with a single bit can produce a great savings in space.

What have seen this semester where using only a single bit to represent something could have produced a substantial savings in space?

Recall how we represented csets:

```
int cset[CSET_SIZE];  // 512 bytes!
```

How much space is actually required to represent a cset?

One bit per character!

```
int cset[CSET_SIZE/32]; // four ints; sixteen bytes
```

What would be a tradeoff with a bit-per-character representation?

Operations would take longer—it takes more instructions to determine the value of a single bit than it takes to determine the value of an **int**.

Bit-level operators

C has several operators that perform bit-level operations on integer values:

- `~` complement (unary)
- `&` bitwise AND
- `|` bitwise OR
- `^` bitwise XOR (exclusive OR)
- `>>` right shift
- `<<` left shift

The complement (~) operator performs a bit by bit complement of its operand's value, changing `0` to `1` and `1` to `0`:

```
short a = 0x0F61; 0000 1111 0110 0001
short b = ~a;     1111 0000 1001 1110
```

The portable way to get a value with all bits "set" is to complement `0`:

```
short c = ~0;     1111 1111 1111 1111
```

Note that **`short`** values are used in these examples but the bit-level operators work on all integer types.

Bit-level operators, continued

The binary `&` operator is bitwise "and". It produces a result bit of `1` if both input bits are `1`. It produces a result bit of `0` otherwise.

```
short a = 0xF0A1; 1111 0000 1010 0001
short b = 0x00F3; 0000 0000 1111 0011
                  -------------------
short c = a & b;  0000 0000 1010 0001
```

The binary | operator is bitwise "or". It produces a result bit of `1` either input bit is `1`. It produces a result bit of `0` otherwise.

```
short a = 0xF0A1; 1111 0000 1010 0001
short b = 0x00F3; 0000 0000 1111 0011
                  -------------------
short c = a | b;  1111 0000 1111 0011
```

Bit-level operators, continued

The binary `^` operator is bitwise "exclusive or". It produces a result bit of `1` if the two input bits <u>differ</u>. It produces a result bit of `0` otherwise.

```
short a = 0xF0A1; 1111 0000 1010 0001
short b = 0x00F3; 0000 0000 1111 0011
                  -------------------
short c = a ^ b;  1111 0000 0101 0010
```

What's the result of `c ^ c`?

```
  1111 0000 0101 0010
^ 1111 0000 0101 0010
  -------------------
  0000 0000 0000 0000
```

On some machines the fastest way to make a zero at the instruction level is to XOR a *register* with itself: `xor r1,r1`

https://randomascii.wordpress.com/2012/12/29/the-surprising-subtleties-of-zeroing-a-register/

Shifts

The binary operators << and >> are left shift and right shift, respectively.

The expression

```
a << 1
```

indicates to shift the bits of a to the left by one bit position and insert a 0 as the rightmost bit. The leftmost bit is discarded. The right-hand operand can be any integer value. The shifted value is the result.

Example with left shift (results are cumulative):

```
short a = 1;   0000 0000 0000 0001
a = a << 1;    0000 0000 0000 0010
a = a << 2;    0000 0000 0000 1000
a = 0xFF << a; 1111 1111 0000 0000
```

Shifts, continued

The right shift operator is `>>`. There are rules about the sign bit for right shifts:

- If the value is unsigned, `0`s are shifted in from the left.
- If the value is signed, `0`s may be shifted in or the sign bit may be propagated; the behavior is compiler-dependent.

Here's a signed `short`:

```
short a = -32768; 1000 0000 0000 0000
a = a >> 1;       1100 0000 0000 0000
a = a >> 4;       1111 1100 0000 0000
```

Contrast with an `unsigned short`:

```
unsigned short a = 0x8000; 1000 0000 0000 0000
               a = a >> 1; 0100 0000 0000 0000
               a = a >> 4; 0000 0100 0000 0000
```

Java has unsigned right shift operator (`>>>`). Why?

Multiplication and division with shifting

A very fast way to divide or multiply by powers of 2 is to shift.

```
char i = 3;     0000 0011
i = i << 1;     0000 0110    // 3 * 2 = 6
i = i << 2;     0001 1000    // 6 * 4 = 24

char i = 12;    0000 1100
i = i >> 1;     0000 0110    // 12 / 2 == 6
i = i >> 2;     0000 0001    // 6 / 4 == 1
```

Some compilers will generate a shift instruction when an integer is being divided or multiplied by a constant that's a power of `2`, such as `i = j / 2`.

Where do bits go when they're shifted off the left- or right-end of a value?

The bit bucket!

How many bits are in an **int**?

Problem: How could we figure out how many bits are in `int`?

```
unsigned int i = ~0;
int nbits = 0;

while (i) {
    nbits++;
    i >>= 1;
    }

printf("%d bits\n", nbits);
```

Example: **first_bit_set**

Here's a routine to find the position of the leftmost "set" bit in an **int**:

```
int first_bit_set(int w)
{
   int intbits = sizeof(int) * CHAR_BIT;
   unsigned int checker = 1 << (intbits - 1);
          // checker = 10000000...00000000
   for (int pos = intbits; pos; pos--)
      if (w & checker)
         return pos;
      else
         checker >>= 1;

   return 0;
}
```

Results:

```
  0 (00): 0
  1 (01): 1
  2 (02): 2
  3 (03): 2
  4 (04): 3
  5 (05): 3
  6 (06): 3
  7 (07): 3
  8 (08): 4
  9 (09): 4
 ...
 13 (0D): 4
 14 (0E): 4
 15 (0F): 4
 16 (10): 5
 17 (11): 5
```

CHAR_BIT, the number of bits in a **char** is in **<limits.h>**.

How does it work?

Another example

A function to convert an **int** to a character string of zeros and ones:

```
void val_to_bits(int val, int size, char *buf)
{
    int nbits = size * CHAR_BIT;
    for (i = nbits - 1; i >= 0; i--) {
        char bit = (val & 1<<i) ? '1' : '0';
        *buf++ = bit;
        if (i % 4 == 0 && i != 0)
            *buf++ = ' ';
        }
    *buf = 0;
}
```

Usage:

```
char buf[sizeof(i)*CHAR_BIT + 1];
val_to_bits(0xCafeBabe, sizeof(int), buf);
printf("%X is %s\n", i, buf);
```

Output:

```
CAFEBABE is 1100 1010 1111 1110 1011 1010 1011 1110
```

Bit fields—sized bit strings

A *bit field* is an integer member of a structure or union whose width is specified in terms of bits. A member name followed by a colon and an integer indicates a bit field.

Bit fields are often used to describe, at the level of the bits, a value directly obtained from hardware, such as status register contents.

Imagine a tape drive error status register with the low eleven bits for a block number, a single bit read/write flag, a three bit error code, and the high bit unused:

```
typedef union {
    unsigned short word;
    struct {
        unsigned block_num: 11;
        unsigned read_write: 1;
        unsigned error_code: 3;
        unsigned : 1; // unnamed bit field
        } parts;
    } tape_error_word;
```

The fields are packed into the minimum amount of space, 16 bits in this case.

Bit fields, continued

Here's a routine to dissect an error code. Note how the **union** is used.

```
void show_tape_error(unsigned short eword)
{
   tape_error_word ew;
   ew.word = eword;

   printf("Tape error: block %u, %s mode, ",
     ew.parts.block_num,
     ew.parts.read_write ? "write":"read");

   printf("code %x\n", ew.parts.error_code);
}
```

For reference:

```
typedef union {
  unsigned short word;
  struct {
    unsigned block_num: 11;
    unsigned read_write: 1;
    unsigned error_code: 3;
    unsigned : 1; // unnamed
    } parts;
 } tape_error_word;
```

For the call **show_tape_error(0x77FF)** this output is produced:

```
Tape error: block 2047, read mode, code 7
```

Note: Bit fields can be used to save space, but there's extra computation involved to isolate the values.

When should we choose bit fields vs. a series of bit-wise operations?

More about the preprocessor

Parameterized macros

Parameters can be specified for a **#define**. Example:

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))
```

Usage:

```
int m1 = MIN(3, -1);

double m2 = MIN(12.34, 10.1);
```

After preprocessing:

```
int m1 = ((3) < (-1) ? (3) : (-1));

double m2 = ((12.34) < (10.1) ? (12.34) : (10.1));
```

Key point: **MIN()** places no constraints on its operands. It works for any **x** and **y** such that **x < y** is valid.

What's a tradeoff with a macro like **MIN**?

There's no function call overhead but there is "code bloat"—each use of **MIN** generates code for the computation.

Parameterized macros, continued

Here are two parameterized macros we've used:

```
char *type;
if (S_ISREG(statbuf.st_mode))
  type = "regular file";
else if (S_ISDIR(statbuf.st_mode))
   type = "directory";
```

Here's what the code above expands to:

```
char *type;
if (((((statbuf.st_mode)) & 0170000) == (0100000)))
   type = "regular file";
else if (((((statbuf.st_mode)) & 0170000) == (0040000)))
   type = "directory";
```

Speculate: Why do macros use so many parentheses?

Parameterized macros, continued

Let's try dropping the parentheses in **MIN**:

```
#define MIN_V1(x,y) ((x) < (y) ? (x) : (y))
#define MIN_V2(x,y) x < y ? x : y

int x = 1, y = 2;
int m1 = 10 + MIN_V1(x + 1, y + 2);
int m2 = 10 + MIN_V2(x + 1, y + 2);

printf("m1 = %d, m2 = %d\n", m1, m2);
```

Output:

```
m1 = 12, m2 = 4
```

The issue is operator precedence. Here's the preprocessor output:

```
int m1 = 10 + ((x + 1) < (y + 2) ? (x + 1) : (y + 2));
int m2 = 10 + x + 1 < y + 2 ? x + 1 : y + 2;
```

A rule when writing macros:

Wrap each parameter and the full expansion in parentheses.

Another pitfall with parameterized macros

Consider this code:

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))

int x = 1, y = 2;
int m = MIN(x++, y++);
printf("m = %d, x = %d, y = %d\n", m, x, y);
```

Output:

```
m = 2, x = 3, y = 3
```

Note the expansion of MIN(x++, y++):

```
int m = ((x++) < (y++) ? (x++) : (y++));
```

A rule when using macros:

Watch out for expressions with side-effects.

Common practice:

Always capitalize macro names, to alert the user that they are using a macro.

Conditional inclusion (a.k.a conditional compilation)

The preprocessor has several directives to cause conditional inclusion of code. One is #ifdef:

```
while ((c = getchar()) != EOF) {
#ifdef DEBUG
        printf("c = '%d' (%c)\n", c, c);
#endif
        f(c);
        }
```

If #define DEBUG has been previously seen, the code bracketed by #ifdef...#endif is processed. If not, that code turns into whitespace.

```
% gcc -E cpp16.c
...
 while ((c = getchar()) != (-1)) {




        f(c);
        }
```

Conditional inclusion, continued

For reference:

```
while ((c = getchar()) != EOF) {
#ifdef DEBUG
        printf("c = '%d' (%c)\n", c, c);
#endif
        f(c);
        }
```

To activate the debugging code we could add **#define DEBUG** earlier in the file but let's use **-D** instead:

```
% gcc -E -DDEBUG cpp16.c
 while ((c = getchar()) != (-1)) {

        printf("c = '%d' (%c)\n", c, c);

        f(c);
        }
```

What's an advantage of **#ifdef ...** over **if (debug) { ... }**?

The debugging code is truly "gone". It doesn't contribute to code size. It has no run-time overhead.

Conditional inclusion, continued

The #if directive allows evaluation of constant integral expressions:

```
#if (AIX_VER > 12) || (SUNOS_VER > 4) || defined(ZFONTS)
    UseEnhancedFonts();
#else
    UseStandardFonts();
#endif
```

An easy way to "comment out" code is to use #if 0:

```
#if 0
    for(... loop over nodes ...) {
        trans_closure(p, nset); /* args?? */
        Hmm...
        }
#endif
```

Note that using #if 0 avoids problems with nested /* ... */ comments.

Examples of conditional inclusion

Conditional compilation is commonly used to accommodate platform-specific situations with a single body of source code. Here are some examples from the GNU **find** source, version 4.4.2:

```
#ifdef HAVE_SYS_TYPES_H
#include <sys/types.h>
#endif
...
#ifdef HAVE_SYS_MNTIO_H
#include <sys/mntio.h>
#endif
...
#ifdef STDC_HEADERS
#include <stdlib.h>
#else
extern int errno;
#endif
```

Examples, continued

```
 if (!options.open_nofollow_available)
   {
#ifdef STAT_MOUNTPOINTS
     init_mounted_dev_list(0);
#endif
   }
  ...

  starting_desc = open (".", O_RDONLY
#if defined O_LARGEFILE
                  |O_LARGEFILE
#endif
                  );
...

 if (TraversingDown == direction)
      {
#ifdef STAT_MOUNTPOINTS
        isfatal = dirchange_is_fatal(specific_what,isfatal,silent,newinfo);
#else
        isfatal = RETRY_IF_SANITY_CHECK_FAILS;
#endif
      }
```

Examples, continued

```
#ifdef AFS
#include <netinet/in.h>
#include <afs/venus.h>
#if __STDC__
/* On SunOS 4, afs/vice.h defines this to rely on a pre-ANSI cpp.  */
#undef _VICEIOCTL
#define _VICEIOCTL(id)  ((unsigned int ) _IOW('V', id, struct ViceIoctl))
#endif
#ifndef _IOW
/* AFS on Solaris 2.3 doesn't get this definition.  */
#include <sys/ioccom.h>
#endif
```

The preprocessor provides great flexibility but creates the potential of creating code that is very hard to reason about.

The designers of Java—expert C programmers—made a very conscious decision to not include a preprocessor in Java.

"Include guards"

Sometimes a file will include two files that in turn include the same file. In the code below, both **Circle.h** and **Line.h** include **Point.h**:

% **cat cpp18.c** #include "Line.h" #include "Circle.h" int main() { ... }	% **cat Circle.h** #include "Point.h" typedef struct { Point center; double radius; } Circle;
% **cat Line.h** #include "Point.h" typedef struct { Point p1, p2; } Line;	% **cat Point.h** typedef struct { double x, y; } Point;

A second definition of a structure, even if identical, is an error:

```
% gcc cpp18.c
In file included from Circle.h:1:0,
                 from cpp18.c:2:
Point.h:3:7: error: conflicting types for 'Point'
Point.h:3:7: note: previous declaration of 'Point' was here
```

"Include guards", continued

The solution is a preprocessor technique known as an *include guard*.

Here's a numbered listing **Point.h** with an include guard:

```
1 #ifndef Point_h_
2
3 #define Point_h_
4
5 typedef struct {
6     double x, y;
7     } Point;
8
9 #endif
```

The first time that **Point.h** is included, no **#define** has been seen for a macro named **Point_h_**. By virtue of that absence, the **#ifndef Point_h_** succeeds and lines 2-8 are processed. Line 3 defines **Point_h_** and lines 5-7 declare a **Point** type.

Things are different the if **Point.h** is included a second time. Because of the previous inclusion of **Point.h** the macro **Point_h_** is defined. The **#ifndef Point_h_** will fail and lines 2-8 will turn into whitespace.

Include guards, continued

Here's what **cpp -E cpp18.c** shows:

```
# 1 "Line.h" 1

# 1 "Point.h" 1

typedef struct {
   double x, y;
   } Point;
# 4 "Line.h" 2
typedef struct {
   Point p1, p2;
   } Line;
# 2 "cpp18.c" 2
# 1 "Circle.h" 1
```

*(Without the include guard, the code from **Point.h** would appear here, too.)*

```
typedef struct {
   Point center;
   double radius;
   } Circle;
# 3 "cpp18.c" 2
int main() { ... }
```

<table>
<tr>
<td>

% **cat cpp18.c**
#include
"Line.h"
#include
"Circle.h"
int main() { ... }

</td>
<td>

% **cat Circle.h**
#ifndef Circle_h
#define Circle_h
#include "Point.h"
typedef struct {
   Point center;
   double radius;
   } Circle;
#endif

</td>
</tr>
<tr>
<td>

% **cat Line.h**
#ifndef Line_h
#define Line_h
#include
"Point.h"
typedef struct {
   Point p1, p2;
   } Line;
#endif

</td>
<td>

% **cat Point.h**
#ifndef Point_h
#define Point_h
typedef struct {
   double x, y;
   } Point;
#endif

</td>
</tr>
</table>

The ## operator

The preprocessor has a ## operator. It concatenates two tokens.

Given:

```
#define JOIN(a,b) a##b
```

then JOIN(Fish,Knuckles) expands to FishKnuckles.

Let's use ## to write a macro that generate typed "min" functions:

```
#define TYPED_MIN(type) \
    type type##_min(type a, type b) { return a < b ? a : b; }
```

With the above, the lines

```
TYPED_MIN(int)
TYPED_MIN(double)
```

expand to:

```
int int_min(int a, int b) { return a < b ? a : b; }
double double_min(double a, double b) { return a < b ? a : b; }
```

Predefined macros

The preprocessor makes available a number of predefined macros. Here are four handy ones: (See C11 6.10.8 for the full list.)

- **__DATE__** Current date (**char ***)
- **__TIME__** Current time (**char ***)
- **__FILE__** Source file name (**char ***)
- **__LINE__** Line number (**int**)

Example:

```
printf("Compiled at %s on %s\n", __TIME__,  __DATE__);
```

After preprocessing:

```
printf("Compiled at %s on %s\n", "22:37:59", "Nov 29 2015");
```

The **assert** macro uses **__LINE__** and **__FILE__**. **assert(0 == 1)** produces output like this:

```
a.out: cpp1.c:13: main: Assertion `0 == 1' failed.
```

Challenge:

Use **__LINE__** and **__FILE__** to generate **alloc_block(...)** and **free_block(...)** calls that include file and line information.

Multiple source files

Multiple source files

Large C programs are typically broken into many source files. Here's a trivial program that's broken into two files:

% cat hello.c	% cat do_hello.c
double version = 1.0;	#include <stdio.h>
void do_hello(const char *);	char *version;
int main(int argc, char **argv)	void do_hello(void)
{	{
do_hello(argv[1]);	printf("Hello, world! (Version %s)\n",version);
}	}

Compilation and execution:

```
% gcc hello.c do_hello.c
% a.out
Segmentation fault (core dumped)
```

Note that the program compiles without warning. What's wrong?

main passes **do_hello** an argument but **do_hello** takes no arguments.

do_hello expects the global **version** to be a string but in **main** it's a **double**.

We need a way to ensure that there are no mismatches between multiple files.

Headers prevent mismatches

Let's introduce a header file that specifies common elements.

```
% cat hello2.h
char *version;
void do_hello(void);
```

New versions, **hello2.c** and **do_hello2.c**:

% cat hello2.c	% cat do_hello2.c

#include "hello2.h"
double version = 2.0;
int main(int argc, char **argv)
{
do_hello(argv[1]);
}
	#include <stdio.h>
	#include "hello2.h"
	void do_hello(void)
	{
	printf("Hello, world! (Version %s)\n", version);
	}

Compilation, with good errors!

```
% gcc hello2.c do_hello2.c
hello2.c:2:8: error: conflicting types for 'version'
hello2.h:1:14: note: previous declaration of 'version' was here
hello2.c: In function 'main':
hello2.c:5:5: error: too many arguments to function 'do_hello'
hello2.h:2:6: note: declared here
```

Headers prevent mismatches, continued

Final version, with all issues resolved:

```
% cat hello3.h
char *version;
void do_hello(void);
```

% cat hello3.c	% cat do_hello3.c

#include "hello3.h"
char *version = "3.0a";
int main()
{
   do_hello();
}
	#include <stdio.h>
	#include "hello3.h"
	void do_hello(void)
	{
	   printf("Hello, world! (Version %s)\n", version);
	}

Compilation and execution:

```
% gcc hello3.c do_hello3.c
% a.out
Hello, world! (Version 3.0a)
```

The moral of the story:

Putting declarations in header files prevent mismatches.

Linking

Recall the steps in producing an executable from a file such as **hello.c**:

Preprocess **hello.c**

Compile the preprocessed C code (a translation unit) into assembly code, **hello.s**.

Using **as**, assemble **hello.s** into an object file, **hello.o**.

Using **ld**, "link" **hello.o** with library functions to produce an executable.

Let's use **nm** to look at the object files produced from **hello3.c** and **do_hello3.c**:

```
% gcc -c hello3.c do_hello3.c
% nm hello3.o
                  U do_hello
0000000000000000  T main
0000000000000000  D version

% nm do_hello3.o
0000000000000000  T do_hello
                  U printf
0000000000000008  C version
```

The "linker", **ld**, needs to make a connection between **hello3.o**'s reference to **do_hello** and **do_hello3.o**'s definition of **do_hello.** It links the reference to the definition.

Similarly, **do_hello3.o**'s reference to **printf** needs to be linked to the definition of **printf** in the C library.

Also, **do_hello3.o**'s reference to **version** needs to be linked to the definition in **hello3.o**.

Linking, continued

Let's run **gcc** on the object files. **gcc** will run **ld** to produce an executable.

```
% gcc -o hello3 hello3.o do_hello3.o
```

Let's run **nm** on the executable. Here are some key lines from **nm**'s output:

```
% nm hello3
 0000000000400410  T _start
0000000000400504  T do_hello
00000000004004f4  T main
                  U printf@@GLIBC_2.2.5
0000000000601020  D version
```

Let's run **gdb** and look for those symbols.

```
% gdb hello3
(gdb) p main
$1 = {int ()} 0x4004f4 <main>
(gdb) p do_hello
$2 = {void (void)} 0x400504 <do_hello>
(gdb) p version
$3 = 0x40061c "3.0a"
(gdb) p &version
$4 = (char **) 0x601020
```

Linkage

The *linkage* of an identifier determines how widely visible it is to other parts of the program.

If an identifier has *external linkage*, it is visible outside the file it is defined in.

By default, function names have external linkage.
do_hello() in **do_hello3.c** can be called from **main()** in **hello3.c** because it (**do_hello**) has external linkage.

By default, the names of global variables have external linkage.
Thus, **version** in **hello3.c** can be used in **do_hello3.c**.

Just like public fields and methods in a Java class can accessed from outside the class, C variable and objects with external linkage can be accessed from outside their containing translation unit.

What's a case where we don't want a function name or global variable name to have external linkage?

Linkage, continued

Let's look at the identifiers with external linkage in **a12/alloc.o**.

```
% nm a12/alloc.o
0000000000000086 T add_pool
000000000000020f T alloc_block
000000000000006c T alloc_error
00000000000007e3 T block_addr
0000000000000000 T block_status
000000000000036e T check_block
00000000000003fa T check_blocks
0000000000000523 T free_block
0000000000000000 B pool_root
000000000000082e T show_pool
000000000000099b T show_pools
0000000000000808 T user_block_addr
```

A user of **alloc.o** can access all of those things, even though the specified interface for the allocator is only **add_pool**, **alloc_block**, **free_block**, **show_pools**, and **check_blocks**. Is that good or bad?

Bad! If an important user starts using any of those helper functions, we might be saddled with supporting the current behavior of those functions.

Internal linkage

The opposite of external linkage is *internal linkage*. If an identifier has internal linkage, it can only be accessed in that file; it is simply not visible elsewhere.

The **static** keyword is (over)used to specify internal linkage. Let's apply it to a couple of things in **alloc.c** that we don't want to expose:

```
static struct pool_info pool_root = {};

static char *block_addr(struct pool_info *pool, int n)
{
   return pool->memory + n * pool->config.real_block_size;
}
```

Attempts to use **pool_root** or **block_addr** from files other than **alloc.c** will produce an "**undefined reference**" error at link-time—they aren't visible outside of **alloc.c**.

What's a rough analogy between internal linkage and an element of Java?

Internal linkage, continued

If we add **static** to everything in **alloc.c** that shouldn't be externally visible, here's what nm shows:

```
% nm a12/alloc.o
0000000000000086 T add_pool
000000000000020f T alloc_block
000000000000006c t alloc_error
00000000000007e3 t block_addr
0000000000000000 t block_status
000000000000036e t check_block
00000000000003fa T check_blocks
0000000000000523 T free_block
0000000000000020 b pool_root
000000000000082e t show_pool
000000000000099b T show_pools
0000000000000808 t user_block_addr
```

What's the difference from the earlier **nm** output?

Lowercase letters are used to indicate internal linkage.

No linkage

Some identifiers have no linkage. Examples:

- Local variables
- Function parameters
- **typedef**s
- **struct**, **union**, and **enum** tags

A key point in understanding linkage is that linkage is an attribute of an identifier.

The **extern** specifier

Here's an under-enforced rule:

There can be only one definition for a variable.

Here's a violation:

% cat hello3.h char *version; void do_hello(void);	% cat hello3.c #include "hello3.h" char *version = "3.0a"; int main() { do_hello(); }

We can see it clearly with **gcc -E**:

```
% gcc -E hello3.c
# 1 "hello3.h" 1
char *version;
void do_hello(void);
# 2 "hello3.c" 2
char *version = "3.0a";
...
```

gcc -c hello3.c produces no complaints.

extern, continued

At hand, **hello3.h**:

```
% cat hello3.h
char *version;
void do_hello(void);
```

By the letter of the standard, we should use **extern** in **hello3.h**:

```
% cat hello3.h
extern char *version;
void do_hello(void);
```

The **extern** says, "Somewhere else there's a **char *** named **version** defined. I want to use it."

You'll often see **extern** on function prototypes, too:

```
extern char *version;
extern void do_hello(void);
```

Try **gcc -E ... | grep extern** to see a lot of **extern**s.

The standard specifies a number of rules about **extern**, but with **gcc**, **extern** is seldom needed.

make

The problem

When the code for a program is distributed among several source files, a simple but workable approach to build the program is a script that does one big compilation:

```
% cat buildit
gcc -o find find.c fstype.c parser.c pred.c tree.c util.c version.c
```

After making a change, the developer runs **buildit**. Everything is recompiled, and the developer tests the new executable.

A "build all" script is simple and nearly foolproof but it becomes increasingly inefficient as the amount of source code grows.

The problem, continued

To avoid unnecessary recompilation the developer might create a script that simply links the object files:

```
% cat linkit
gcc -o find find.o fstype.o parser.o pred.o tree.o util.o version.o
```

After a change, the developer recompiles modified files by hand and then uses **linkit** to create an executable:

```
% vim util.c
% gcc -c util.c
% linkit
```

However, there's the possibility of a forgotten change:

```
% vi parser.c
[...Phone call, meeting, lunch...]
% vi pred.c fstype.c
% gcc -c pred.c fstype.c
% linkit
[...Tests and wonders why parser changes aren't working...]
```

The problem, continued

Another common error is changing a header file but not recompiling all files that include that header file:

```
% vi pred.h
% gcc -c pred.c util.c
```
(Forgot that parser.c*, too, includes* pred.h*.)*

```
% linkit
```
[...Tests and wonders why there's a core dump in parser...]

A solution: `make`

With C, there's a chain of dependencies:

- Executables are built from object files (`.o` files)
- Object files are built from `.c` files
- `.c` files depend on `.h` files
- With a *parser generator*, `.c` and `.h` files might be created from a grammar.

With `make`, we specify:

- Dependencies between files
- How to create files based on existing files

Based on those specifications, `make` will take appropriate steps to build whatever we request.

`make` has built-in rules for many languages, including C, but rules can be added to enable `make` to build many types of things, like documents.

make basics

The operation of make is controlled by a "makefile".

Here's a simple makefile, named **Makefile**:

```
% cat Makefile
CC=gcc
hello: hello.o do_hello.o
	gcc -o hello hello.o do_hello.o     # NOTE: TAB at start of line!
```

The first line sets the value of the **make** variable **CC**. **CC** specifies the command to use for compiling C files. It defaults to **cc** but we want to use **gcc**.

The second line describes a *target* named **hello**. The target's name appears to the left of the colon. The names following the colon are the files that the target "depends" on. In essence it specifies this:

"Before creating **hello**, be sure that **hello.o** and **do_hello.o** are up to date."

The third line specifies how to create **hello**, assuming **hello.o** and **do_hello.o** are up to date.

make basics, continued

The makefile at hand:

```
CC=gcc
hello: hello.o do_hello.o
	gcc -o hello hello.o do_hello.o
```

Again, the second line says that before creating **hello**, **hello.o** and **do_hello.o** must be up to date.

make has a built-in rule that says that **.o** files depend on **.c** files: If **hello.o** is needed, **make** looks for **hello.c**

A **.o** file is considered to be up to date if the **.o** file is newer than the corresponding **.c** file. If the **.o** file is older or does not exist, the **.o** file must be (re)created.

To create a **.o** file from a **.c** file, **make** uses a built-in rule: Compile the **.c** file with the program specified by **CC**.

make basics, continued

The makefile at hand:

```
CC=gcc
hello: hello.o do_hello.o
	gcc -o hello hello.o do_hello.o
```

If we want **make** to build a target described in a makefile, we invoke **make** with the target name. Here is **make** in action:

```
% ls
Makefile  do_hello.c  hello.c  hello.h

% make hello
gcc    -c -o hello.o hello.c
gcc    -c -o do_hello.o do_hello.c
gcc -o hello hello.o do_hello.o

% ls -lt
total 22
-rwxrwxr-x 1 whm whm 8487 Dec  2 00:54 hello
-rw-rw-r-- 1 whm whm 1592 Dec  2 00:54 do_hello.o
-rw-rw-r-- 1 whm whm 1584 Dec  2 00:54 hello.o
-rw-rw-r-- 1 whm whm  116 Dec  2 00:31 do_hello.c
-rw-rw-r-- 1 whm whm   74 Dec  2 00:31 hello.c
-rw-rw-r-- 1 whm whm   36 Dec  2 00:30 hello.h
-rw-rw-r-- 1 whm whm   94 Dec  2 00:30 Makefile
```

make basics, continued

```
CC=gcc
hello: hello.o do_hello.o
   gcc -o hello hello.o do_hello.o
```

Note the result of a second **make** command:

```
% make hello
make: 'hello' is up to date.
```

hello is declared to be up to date because it is newer than **hello.o** and **do_hello.o**, the two files on which it depends, and in turn, **hello.o** and **do_hello.o** are newer than **hello.c** and **do_hello.c**, respectively.

Note the result of a **make** after "changing" **hello.c** by **touch**ing it.

```
% touch hello.c
% make hello
gcc    -c -o hello.o hello.c
gcc -o hello hello.o do_hello.o
```

Due to the **touch**, **hello.o** was older than **hello.c** and therefore **hello.o** was rebuilt. After that, **hello** was older than **hello.o** and therefore **hello** was rebuilt.

<u>It is important to understand that **make** bases its actions solely on modification times, not file contents.</u>

DRY!

Note that "hello.o do_hello.o" appears twice:

```
CC=gcc
hello: hello.o do_hello.o
	gcc -o hello hello.o do_hello.o
```

Let's introduce a variable for the .o files:

```
OBJS=hello.o do_hello.o
hello: $(OBJS)
	gcc -o hello $(OBJS)
```

A **clean** target

A common convention is a **clean** target, to delete everything that can be recreated with a **make**.

```
CC=gcc
OBJS=hello.o do_hello.o
hello: $(OBJS)
	gcc -o hello $(OBJS)

.PHONY: clean
clean:
	rm hello $(OBJS)
```

Usage:

```
% make clean
rm -f hello hello.o do_hello.o
```

Speculate: What's the purpose of **.PHONY**?

If we accidentally created a file named **clean**, then **make clean** would just say "**make: 'clean' is up to date.**"

A **backup** target

Here's a target to make a backup with **mar** and mail it to oneself:

```
backup:
    @echo Making archive...
    @mkdir -p .backups
    @mar c .backups/backup-$$(date "+%Y%m%d.%H%M").mar  \
        Makefile *.c
    @echo Mailing...
    @mail -s "backup of hello" $(USER) < $$(ls -t .backups/* | head -1)
    @echo Done!
    @echo Most recent backups:
    @ls -ltr .backups | tail -3
```

Notes:

make won't echo commands that start with **@**.
$$** is used to get a single **$ through to the shell.
$(USER) uses the **USER** environment variable from the shell.
backup should be added to **.PHONY** list.
A backslash is used to break the **mar** command across two lines.

Rather than putting all those commands in the makefile should we make a script with those lines and have the **backup** target run it?

A **backup** target, continued

For reference:

```
backup:
    @echo Making archive...
    @mkdir -p .backups
    @mar c .backups/backup-$$(date "+%Y%m%d.%H%M").mar  \
        Makefile *.c
    @echo Mailing...
    @mail -s "backup of hello" $(USER) < $$(ls -t .backups/* | head -1)
    @echo Done!
    @echo Most recent backups:
    @ls -ltr .backups | tail -3
```

Execution:

```
% make backup
Making archive...
Added Makefile
...more mar output...
Mailing...
Done!
Most recent backups:
-rw-rw-r-- 1 whm whm 787 Dec  3 18:17 backup-20151203.1817.mar
...
```

Execution of multiple targets

Let's add a simple testing target.

```
test: hello
	@echo Testing...
	./hello
```

Note that **test** depends on **hello**. Also, **test** is another phony target—we never make a file named **test**, so we should add **test** to **.PHONY**

Let's get a clean slate, then build and test:

```
% make clean hello test
rm -f hello *.o
gcc    -c -o hello.o hello.c
gcc    -c -o do_hello.o do_hello.c
gcc -o hello hello.o do_hello.o
Testing...
./hello
Hello, world! (Version 3.0a)
```

Header dependencies

Here is **hello.c**:

```
#include "hello.h"
char *version = "3.0a";
int main()
{
   do_hello();
}
```

Let's move **version** into a new header file:

```
% cat version.h
char *version = "3.0a";

% cat hello.c
#include "hello.h"
#include "version.h"
int main()
...
```

Header dependencies, continued

Let's do a build...

```
% make hello
gcc    -c -o hello.o hello.c
gcc    -c -o do_hello.o do_hello.c
gcc -o hello hello.o do_hello.o
```

Let's use **echo** to simulate an edit of the version number...

```
% echo 'char *version = "4.0";' > version.h
```

Let's test the latest...

```
% make test
Testing...
./hello
Hello, world! (Version 3.0a)
```

Why did we get the old version number? The target **test** does depend on **hello** being up-to-date:

```
test: hello
        @echo Testing...
        ./hello
```

Header dependencies, continued

The situation: We updated **version.h** but **make test** ran an old version. Here's the involved code from the makefile:

```
OBJS=hello.o do_hello.o
hello: $(OBJS)
	gcc -o hello $(OBJS)

test: hello
	@echo Testing...
	./hello
```

What's wrong?

Here's **hello.c**:

```
#include "hello.h"
#include "version.h"
int main()
...
```

hello.c uses **hello.h** and **version.h** but the makefile has no reflection of that dependency. It's a makefile bug!

Header dependencies, continued

make's built-in dependency of **.o** files on **.c** files is insufficient. We need to add explicit dependencies for both **do_hello.o** and **hello.o**:

```
do_hello.o: do_hello.c hello.h          # Added
hello.o: hello.c hello.h version.h      #   " "

hello: $(OBJS)
	gcc -o hello $(OBJS)

test: hello
	@echo Testing...
	./hello
```

Let's try **test** again:

```
% make test
gcc    -c -o hello.o hello.c
gcc -o hello hello.o do_hello.o
Testing...
./hello
Hello, world! (Version 4.0)
```

Header dependencies, continued

It's tedious and error prone to manually maintain header file dependencies in a makefile. One option is to use gcc -MM to compute the dependencies for us:

```
% gcc -MM *.c
do_hello.o: do_hello.c hello.h
hello.o: hello.c hello.h version.h
```

However, we still need to update the makefile with that output.

A simpler approach for header file dependencies that works well for small projects is to specify that all object files depend on all header files:

```
hello: $(OBJS)
	gcc -o hello $(OBJS)

*.o: *.h # Added, and above .o dependencies removed
```

Usage:

```
% touch version.h
% make hello
gcc    -c -o hello.o hello.c
gcc    -c -o do_hello.o do_hello.c
gcc -o hello hello.o do_hello.o
```

The $@ variable

When a target is being built we can access "automatic" variables that hold information about the target. One is $@, which is the name of the current target.

Here's a trivial example:

```
abc:
        @echo The target is $@

xyz:
        @echo The target is $@
```

Execution:

```
% make abc
The target is abc

% make xyz
The target is xyz
```

The **$@** variable, continued

Repetitious specification of targets can often be avoided by using **$@**.

Here's a makefile that builds **vector[123]** if **vector.c** is changed:

```
CC=gcc
PROGS=vector1 vector2 vector3

all: $(PROGS)

$(PROGS): vector.c
	gcc -o $@ $@.c vector.c
```

Usage:

```
% make all
gcc -o vector1 vector1.c vector.c
gcc -o vector2 vector2.c vector.c
gcc -o vector3 vector3.c vector.c
```

Note that **$(PROGS): vector.c** states that all the elements of **PROGS** depend on **vector.c**.

A portion of a12/Makefile

```
.SECONDEXPANSION:
CC=gcc
CFLAGS=-Werror -Wall -g -std=gnu1x -I/cs/www/classes/cs352/fall15/h

PROGS=lastmod kd vm
EXECS=$(PROGS) $(VECTORN)
ALL=$(EXECS) vector.o
VECTORN=vector1 vector2 vector3 vector_ph vector_ps vector_yw vector_zn

all: $(EXECS)

$(VECTORN): a12/$$@.c a12/vector.h vector.c
	$(CC) $(CFLAGS) -o $@ a12/$@.c vector.c

.PHONY: vector
vector: $(VECTORN)
```

Notes:

- **.SECONDEXPANSION** enables **a12/$$@.c** in **$(VECTORN)**
- **CFLAGS** specifies options for **gcc**.
- Target **all** causes all executables for all problems to be built.
- **vector** is a phony target that causes the **vector** executables to be built.

The rest of `a12/Makefile`

```
...
PROGS=lastmod kd vm
EXECS=$(PROGS) $(VECTORN)
ALL=$(EXECS) vector.o

...
all: $(EXECS)
...
$(PROGS): $$@.c
	$(CC) $(CFLAGS) -o $@ $@.c

install: $(ALL)
	chmod 711 $(EXECS)
	cp -i $(EXECS) vector.o a12

clean:
	rm -f *.o $(ALL) *.build core
```

make "the works"

If **make** is run with no arguments, it builds the first target in **Makefile**. Recall that **all** is the first target in a12/Makefile:

```
PROGS=lastmod kd vm
EXECS=$(PROGS) $(VECTORN)
VECTORN=vector1 vector2 vector3 ... vector_zn
all: $(EXECS)
...
```

It follows a common convention: **make** with no arguments builds "the works".

Handy **make** options

By default, **make** looks for **GNUmakefile**, **makefile**, and **Makefile**, in that order, and uses the first one it finds.

The **-f** option can be used to name a specific makefile:

```
make -f Makefile.old
```

The **-n** option indicates to show the commands that need to be performed, but not actually run them.

By default, **make** stops on the first compilation error. **make -k** tells it to keep going. Emacs' **compile** command uses **make -k** by default.

The **-p** option causes **make** to dump out its rules and more. Here's a way to run it with a clean slate, by using **-f** to name **/dev/null** as the makefile:

```
% make -p -f /dev/null > p.out
make: *** No targets.  Stop.
```

Try it, edit **p.out**, and look for occurrences of **COMPILE.c**.

The #1 makefile error

Here is the general form of a target specification:

target1 ... targetN: dependency1 .. dependencyN
<TAB>*command1*
<TAB>...
<TAB>*commandN*

The most common makefile error is to not precede commands with a TAB!

Example:

```
% cat Makefile
test:
        @echo testing...
% make
makefile:2: *** missing separator.  Stop.
```

The problem is that on the second line, spaces, not a TAB precede the @:

```
% cat -A Makefile
test:$
        @echo testing...$
```

cat -A should show this for that second line:

```
^I@echo testing...$
```

Lots more with make

This material provides an introduction to make but is far from comprehensive.

An excellent source of documentation for make is the manual for GNU make:
https://www.gnu.org/software/make/manual/

make was developed in the late 1970s by Stu Feldman at Bell Labs. There are a number of variants and "replacements" for make but none have been as successful on UNIX as make itself. This material is largely based on "classic" make but does introduce elements of GNU make.

Data point: The makefile for bash 4.2 is 1,529 lines.

Ant, Maven, and Gradle are popular build tools for Java applications.

The task of creating and maintaining a build system can be a full-time job. Large projects, especially those targeting multiple platforms, sometimes have "build engineers".

See also: https://en.wikipedia.org/wiki/List_of_build_automation_software

Field Trip!