# Table of Contents

1. **Introduction** ................................................. 1-1  
   - What You’ll Learn 1-2  
   - Survival Tips 1-3  
   - Course Information 1-4  
   - Surfing the Web 1-5

2. **An Introduction to C** .................................... 2-1  
   - Dissecting hello.c 2-2  
   - Computing the Sum from 1 to n 2-3  
   - Dissecting sum.c 2-4  
   - Expression Evaluation 2-5  
   - Another Example: Printing a Random Pattern 2-6  
   - Dissecting pattern.c 2-7  
   - For More Information 2-9

3. **More About C** .............................................. 3-1  
   - Types 3-2  
   - Constants 3-3  
   - Variables 3-4  
   - Expressions 3-5  
   - Precedence and Associativity 3-6  
   - Assignments 3-7  
   - Increment/Decrement 3-8  
   - Idiomatic C 3-9  
   - Statements 3-10  
   - Conditional Expressions 3-12

4. **Functions and Modules** .............................. 4-1  
   - Computing ex 4-2  
   - Dissecting ex.c 4-4  
   - Scope (a.k.a. Visibility) 4-6  
   - Arguments and Locals 4-8  
   - Global Variables 4-9  
   - Modules 4-10

5. **Arrays** ...................................................... 5-1  
   - Printing a Histogram 5-2  
   - Dissecting hist.c 5-3  
   - Multidimensional Arrays 5-4  
   - Printing a Stem-and-Leaf Plot 5-5  
   - Dissecting stem.c 5-6  
   - Passing Arrays to Functions 5-7

6. **Strings** ...................................................... 6-1  
   - Printing Repeated Words 6-2  
   - Dissecting double.c 6-3  
   - Implementing String Handling Functions 6-4  
   - Arrays of Strings 6-5  
   - Dissecting shuffle.c 6-6  
   - Command-Line Arguments 6-7  
   - Testing random() 6-8

7. **The TOY Machine** ....................................... 7-1  
   - Inside the Box 7-2  
   - Memory 7-3  
   - Basic Cycle 7-4  
   - Digression: Number Systems 7-5  
   - Conversions 7-6  
   - Boolean Functions 7-7  
   - Machine Arithmetic 7-8  
   - Two’s-Complement Arithmetic 7-9
17. Analysis of Algorithms .......................... 17-1
   Sublist Sum Problem 17-2
   A Simple Brute-Force Solution 17-3
   Profiling 17-4
   A Better Algorithm 17-5
   Profiling the Better Algorithm 17-6
   The Optimal Algorithm 17-7
   Summary 17-8

18. Elementary Systems Programming ............... 18-1
   Compilation Pipeline 18-2
   Linking 18-4
   Loading 18-5
   Assembly Language 18-6
   Programming in TAL 18-7
   Object Code 18-8
   Linking 18-10
   Loading 18-12
   Separate Compilation 18-13

19. Compilers ........................................... 19-1
   Lexical Analysis 19-2
   Syntax Analysis 19-3
   Parsers 19-4
   Code Generation 19-5
   A Simple Compiler 19-6
   Lexical Analysis 19-7
   Parsing 19-8
   Reverse Polish Notation 19-9
   Code Generation 19-10
   The Main Program 19-11

20. Operating Systems ................................. 20-1
   Multiprogramming 20-2
   Reentrant Programs 20-3
   Virtual Memory 20-4
   Paging 20-5
   Size of Virtual Memory 20-7
   File Systems 20-8
   UNIX File System 20-9
   File Layout 20-10
   Typical Medium-Size File 20-11

21. Regular Expressions ................................ 21-1
   egrep 21-2
   Formal Languages 21-5
   Finite State Automata 21-6
   ‘Bounce’ Filter 21-7
   Simulating FSAs 21-8
   FSAs Can’t ‘Count’ 21-9

22. Hard Problems ...................................... 22-1
   The Traveling Skibum Problem 22-2
   Unsolvable Problems 22-3
   The Halting Problem 22-4
   More Integers or Reals? 22-6
   Implications 22-7

23. Viruses and Secret Messages ..................... 23-1
   Infection Routes 23-2
   Detecting Viruses 23-3
   Fingerprints 23-4
   Fingerprints on the Internet 23-5
   Cryptography 23-6
   Public-Key Cryptosystems 23-8
   RSA Public-Key Cryptosystem 23-9
   RSA Decryption 23-11
   PGP 23-12
Lecture 1. Introduction

• What is ‘computer science?’
  1. The science of manipulating ‘information’
  2. Designing and building systems that do (1);
     e.g., computers, software, networks, ...

• What does COS 126 cover?
  Science (the elegant ideas):
    algorithms and algorithm design (recursion, efficiency, data structures)
    theory of computation (what is computable, intrinsically ‘hard’ problems)
  Engineering (the nuts and bolts):
    programming (the C programming language, machine language)
    basic computer architecture (instruction sets)
    software systems (operating systems, virtual memory, compilers)

• A better name for ‘Computer Science’ might be ‘Computing’
  “Any field that calls itself a science probably isn’t one.”
   — anonymous?
What You’ll Learn

• Just enough to make you dangerous…

• Science:
  how to design algorithms to solve specific problems
  how to choose efficient data structures and algorithms
  how and when to use recursion
  how to recognize hard problems

• Engineering:
  how to write small applications in the C programming language
  how to use C pointers to build dynamic data structures
  how to build a program from smaller subprograms
  how to write assembly language programs
  how programs in high-level languages are translated into machine language
  how to use the UNIX operating system and its tools
  how to browse the World Wide Web

• COS 126 is about computer science, not about getting a job, but 126 will help…
Survival Tips

• Attend lectures and classes
• Go to a ‘Getting Started’ session
• Do the reading; cruise the books on reserve (at the Engineering Library, EQuad)
• Do the exercises; understand the solutions
• Visit the COS 126 Help! Web page when you have questions
• Browse the COS 126 Web often; visit ‘What’s New’ perhaps daily
• Do the programming assignments
• Digest programming assignments as soon as they appear on the COS 126 Web
• Start on programming assignments *early*
• Think before you write code; compose first, then write code
• Use the lab undergraduate teaching assistants
• Ask for help — as soon as you need it!
Course Information

• Nearly all COS 126 material is available only on the World Wide Web
  
  detailed course information (grading, policies, etc.)
  lecture slides (buy the paper copy, too)
  course schedule
  programming assignments
  exercises
  helpful information
  frequently asked questions
  etc.

Exceptions: first handout (how to browse the Web)
exams (two evening midterms, final)
perhaps a few ‘crib’ sheets

• You will submit all assignments electronically; timestamps will tell us when

• You are responsible for getting the necessary material and meeting deadlines

• Save trees — don’t print Web pages unnecessarily
Surfing the Web

• use **netscape** (or another Web browser) to access course materials

  % **netscape** http://www.cs.princeton.edu/courses/cs126/ & ↵

  slanted font indicates what you type; ↵ denotes the ‘enter’ or ‘new-line’ key

• The course URL — universal resource locator — is

  http://www.cs.princeton.edu/courses/cs126/

  You can browse the course Web from **anywhere**, if you have computer and
  Internet access (e.g. America Online)
Lecture 2. An Introduction to C

• Everyone’s first C program: hello.c

```
/* Everyone’s first
 C program. */
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

• To compile, load, and execute hello.c:

```
% lcc hello.c
% a.out
Hello world!
%
```

slanted font indicates what you type

• Writing and running C programs involves at least 3 steps:
  1. Using an editor (emacs) to create a file that contains the program (hello.c)
  2. Using a compiler (lcc) to translate the program from C to ‘machine language’
  3. Issuing a command (a.out) to execute the machine-language program

Usually — OK, always — you iterate these steps until step 3 is ‘correct’
Dissecting hello.c
/* Everyone’s first
   C program. */

/* and */ enclose comments, which document your program or parts of it. The
compiler treats a comment as a single space

#include <stdio.h>

#include is a preprocessor directive, which causes the compiler to read in
standard declarations from the header file stdio.h

int main(void) {

Introduces the main function, which is where execution begins. int is the type
of the value returned by main, void indicates that main has no arguments, and
the { begins the body of the function

printf("Hello world!\n");

Calls the standard library function printf, which prints the characters in its
string argument. \n is an escape sequence for a new-line character

return 0;

main returns the integer 0, indicating that the program completed successfully

}  

Ends the function main
Computing the Sum from 1 to n

/*
Compute the sum of the integers
from 1 to n, for a given n.
*/
#include <stdio.h>

int main(void) {
    int i, n, sum;
    sum = 0;
    printf("Enter n:
");
    scanf("%d", &n);
    i = 1;
    while (i <= n) {
        sum = sum + i;
        i = i + 1;
    }
    printf("Sum from 1 to %d = %d\n", n, sum);
    return 0;
}

% lcc sum.c
% a.out
Enter n:
100
Sum from 1 to 100 = 5050
%
Dissecting sum.c

int i, n, sum;

This *declaration* introduces three *variables* that can store integers — values of type *int*

sum = 0;

This *assignment expression* changes the value stored in sum to 0

scanf("%d", &n);

Calls the *standard library* function *scanf* to read an integer (*%d*) and store it in *n*

i = 1;

Changes the value stored in *i* to 1

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

This *while loop* executes the loop body — the two statements between { and } — repeatedly as long as the value of *i* is less than or equal to the value of *n*
Expression Evaluation

sum = sum + i;

This assignment expression means:

- add the value of sum to the value of i, then
- store that result back into the variable sum

The meaning of this assignment — its semantics — might be clearer if written as

\[ \text{sum + i --> sum;} \]

but that’s not C (or any other language)

\[ i = i + 1; \]

Stores the sum of i and 1 back into i — increments i by 1

\[ \text{printf("Sum from 1 to } \%d \text{ = } \%d; \text{\n", n, sum);} \]

Calls printf to output its first argument; each conversion specifier \%d causes the value of the corresponding following int argument to be printed instead

\[ \text{printf("Sum from 1 to } \%d \text{ = } \%d; \text{\n", n, sum);} \]

\[ \]
Another Example: Printing a Random Pattern

```c
/*
Print a N x N random pattern.
*/
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int i, j, n, bit;
    scanf("%d", &n);
    for (i = 0; i < n; i = i + 1) {
        for (j = 0; j < n; j = j + 1) {
            bit = (rand() >> 14) % 2;
            if (bit == 0)
                printf(" ");
            else
                printf("*");  
        }
        printf("\n");
    }
    printf("\n");
    return 0;
}
```

% lcc pattern.c
% a.out
20
*   **   *    *    *    *    *    *    *    *    *    *    *    *    *    *  
*   *   **    *    *    *    *    *    *    *    *    *    *    *    *    *    *  
*   *   *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *  
*   *   **    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *  
*   *   *    *    *    *    *    *    *    *  
*   *   **    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *  
*   *   **    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *  
*   *   **    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *  
*   *   *    *    *    *    *    *    *    *  
*   *   *    *    *    *    *    *    *  

%
Dissecting pattern.c

for (i = 0; i < n; i = i + 1) {
    ...
}

This for loop executes its body (...) \(n\) times; it is equivalent to

\[
i = 0;\\
while (i < n) {\\
    ...\\
    i = i + 1;\\
}
\]

for (i = 0; i < n; i = i + 1) {
    for (j = 0; j < n; j = j + 1) {
        ...
    }
}

These two nested for loops execute the body of the inner loop \(n \times n = n^2\) times
Dissecting pattern.c, cont’d

\[ \text{bit} = (\text{rand()}} >> 14) \mod 2; \]

This assignment expression

calls the standard function \text{rand}, which returns a 15-bit \text{random number},

shifts that number right by 14 bits,

computes the \text{remainder} of dividing that number by 2;

so, \text{bit} is assigned 0 or 1

\[
\text{if (bit == 0)} \\
\quad \text{printf(" ");} \\
\text{else} \\
\quad \text{printf("*");}
\]

This \text{if-else statement} compares \text{bit} with 0;

it prints a space if \text{bit} is equal to 0, or an asterisk if \text{bit} is not equal to 0
For More Information

- Check out the other texts on C programming (on reserve in the Eng. Library):
  Kelley and Pohl, *C by Dissection: The Essentials of C Programming*, 3/e
  Roberts, *The Art and Science of C: An Introduction to Computer Science*

- Check out the reference books (on reserve):
  Harbison and Steele, *C: A Reference Manual*, 4/e
  Kernighan and Ritchie, *The C Programming Language*, 2/e

- Cruise the sample programs on the COS 126 Help! Web page:
  follow the ‘Sample Programs’ link to hello.c, sum.c, etc.
Lecture 3. More About C

• Programming languages have their lingo

• Programming *language*

  Types are ‘categories’ of values int, float, char
  Constants are values of basic types 0, 123.6, "Hello"
  Variables name locations that hold values i, sum
  Expressions compute values/change variables sum = sum + i
  Statements control a program’s *flow of control* while, for, if–else
  Functions encapsulate statements main
  Modules collections of related variables & functions a.k.a. ‘compilation units’

• Programming *environment*

  Text editor (emacs, vi, sam)
  Compiler (lcc, cc, gcc)
  Linker/loader (ld); used rarely, because lcc runs it
  Debugger (gdb)
Types

• A **type** determines
  a set of **values**, and
  what **operations** can be performed on those values

• **Scalar** types
  - `char` a ‘character’; typically a ‘byte’ — 8 bits
  - `int` a signed integer; typically values from −2147483648 to 2147483647
  - `unsigned` an unsigned integer; typically values from 0 to 4294967295
  - `float` single-precision floating point
  - `double` double-precision floating point

• **Pointer** types: *much* more later…

• **Aggregate** types: values that have **elements** or **fields**, e.g., arrays, structures
### Constants

- **Constant values of the scalar types**

  **char**
  - ’a’ character constant (use single quotes)
  - ‘\035’ character code 35 octal, or base 8
  - ‘\x29’ character code 29 hexadecimal, or base 16
  - ‘\t’ tab (‘\011’, do ‘man ascii’ for details)
  - ‘\n’ newline (‘\012’)
  - ‘\’  single quote
  - ‘\’’  single quote
  - ‘\b’  backspace (‘\010’)
  - ‘\0’  null character; i.e., the character with code 0

  **int**
  - 156 decimal (base 10) constant
  - 0234 octal (base 8)
  - 0x9c hexadecimal (base 16)

  **unsigned**
  - 156U decimal
  - 0234U octal
  - 0x9cU hexadecimal

  **float**
  - 15.6F
  - 1.56e1F

  **double**
  - 15.6 ‘plain’ floating point constants are doubles
  - 1.56E1L
Variables

• A variable is the name of a location in memory that can hold values

```c
int i, sum;
float average;
unsigned count;
```

```c
i = 8;
sum = -456;
count = 101U;
average = 34.5;
```

• A variable has a type; it can hold only values of that type

```c
```

• Assignments change the values of variables

```c
sum = sum + i;
```

changes the value of `sum` to -448

• Variables must be initialized before they are used

```c
#include <stdio.h>

int main(void) {
    int x;
    printf("x = %d\n", x);  // output is undefined!
    return 0;
}
```
Expressions

• Expressions use the values of variables and constants to compute new values

• Binary arithmetic operators take two operands produce one result
  
  +    −    addition, subtraction  
  *    /    multiplication, division  
  %    remainder (a.k.a. modulus)

• Type of result depends on type of operands
  
  int i; unsigned u; float f;

<table>
<thead>
<tr>
<th>+</th>
<th>i</th>
<th>u</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>int</td>
<td>unsigned</td>
<td>float</td>
</tr>
<tr>
<td>u</td>
<td>?</td>
<td>unsigned</td>
<td>float</td>
</tr>
<tr>
<td>f</td>
<td>?</td>
<td>?</td>
<td>float</td>
</tr>
</tbody>
</table>

  i + i specifies int addition and yields an int result
  
  int and unsigned division truncate: 7/2 is 3, but 7.0/2 is 3.5

• Unary operators take one operand and produce one result

  −    +    negation, ‘affirmation’ (just returns its operand’s value)
Precedence and Associativity

- Operator precedence and associativity dictate the order of expression evaluation.

  - **Precedence** dictates which subexpressions get evaluated first.
    - Highest unary: - +
    - Binary: * / %
    - Lowest binary: + -
    - \(-2a + b\) is evaluated as if written as \((((-2)a) + b)\)

  - **Associativity** dictates the evaluation order for expressions with several operators of the same precedence.
    - All arithmetic operators have left-to-right associativity.
      - \(a + b + c\) is evaluated as if written as \(((a + b) + c)\)

- Use **parentheses** to force a specific order of evaluation.
  - \(-2(a + b)\) computes \(-2\)
    - the product of these two values
Assignments

• Assignment expressions store values in variables

\[ \text{variable} = \text{expression} \]

the type of expression must be

the same as the type of variable
convertible to the type of variable

int i; unsigned u; float f;

<table>
<thead>
<tr>
<th>=</th>
<th>i</th>
<th>u</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>u</td>
<td>unsigned</td>
<td>unsigned</td>
<td>unsigned</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>

• Augmented assignments combine a binary operator with assignment

\[ \text{variable} += \text{expression} \]
\[ \text{variable} -= \text{expression} \]
...

\[ \text{sum} += i \quad \text{is the same as} \quad \text{sum} = \text{sum} + i \]
Increment/Decrement

• Prefix and postfix operators ++ -- increment and decrement operand by 1
  
  ++n adds 1 to n
  --n subtracts 1 from n

• **Prefix** operator increments operand *before* returning the *new* value
  
  n = 5;
  x = ++n;
  x is 6, n is 6

• **Postfix** operator increments operand *after* returning the *old* value
  
  n = 5;
  x = n++;
  x is 5, n is 6

• Operands of ++ and -- must be **variables**
  
  ++1
  2 + 3++
  are illegal
Idiomatic C

- `sum.c` (in `sum2.c`) rewritten using common idioms involving `+=` and `++`

```c
/*
 Compute the sum of the integers
 from 1 to n, for a given n.
 */
#include <stdio.h>

int main(void) {
    int i, n, sum = 0;
    printf("Enter n:\n");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        sum += i;
    printf("Sum from 1 to %d = %d\n", n, sum);
    return 0;
}
```

- `scanf` is a form of assignment; it `changes` `n`
Statements

• Expression statements

  \[expression_{\text{opt}};\quad \text{sum} += \text{i};\]
  \[
  \text{printf("Sum from 1 to } \%d = \%d\n", \text{n}, \text{sum});}

• Selection statements

  \[
  \text{if ( conditional ) statement}
  \]
  \[
  \text{if ( conditional ) statement else statement}
  \]
  \[
  \text{if (x > max) max = x;}
  \]
  \[
  \text{if (bit == 0) printf(" "); else printf("*");}\]
  \[
  \text{switch ( expression ) \{ case constant : statement... default : statement } \}

• Iteration statements (loops)

  \[
  \text{while ( conditional ) statement}
  \]
  \[
  \text{while (i <= n) \{ sum += i; i++; } \}
  \]
  \[
  \text{for ( expression_{\text{opt}}; conditional_{\text{opt}}; expression_{\text{opt}}) statement}
  \]
  \[
  \text{for (i = 1; i <= n; i++) sum += i;}
  \]
  \[
  \text{for (;;) printf("Help! I’m looping\n");}
  \]
  \[
  \text{do statement while ( expression ) ;}
  \]
  \[
  \text{do \{ sum += i; ++i; } \while (i <= n);}
Statements, cont’d

• Compound statements

  { declaration_{opt}... statement... }

  for (j = 0; j < n; j = j + 1) {
    int bit = (rand()>>14)%2;
    if (bit == 0)
      printf(" ");
    else
      printf("*");
  }

• Others

  return expression_{opt} ;

  return;

  return 0;

  return -2*(a + b);

  break ;

  continue ;

• Keywords (if else while do for switch case ...) cannot be used as variables
Conditional Expressions

• A **conditional** expression is *any* expression that evaluates to zero or nonzero

• There is no ‘Boolean’ type; nonzero is true, zero is false

• Relational operators compare two arithmetic values (or pointers) and yield 0 or 1

  - `<`  \( \leq \)  less than, less than or equal to
  - `==`  `!=`  equal to, not equal to
  - `>`  \( \geq \)  greater than, greater than or equal to

• Logical connectives

  \[ \text{conditional}_1 \quad \&\& \quad \text{conditional}_2 \quad \begin{cases} 1 \text{ if both conditionals are nonzero} \\ 0 \text{ otherwise} \end{cases} \]

  \[ \text{conditional}_1 \quad |\| \quad \text{conditional}_2 \quad \begin{cases} 1 \text{ if either conditional is nonzero} \\ 0 \text{ otherwise} \end{cases} \]

  **conditionals** are evaluated left-to-right *only as far as is necessary*:

  - `&&`  stops when the outcome is known to be zero
  - `||`  stops when the outcome is known to be nonzero

• Associativity: left to right; precedence: below the arithmetic operators

  \[ \begin{array}{r}
  \text{highest} \quad \text{arithmetic operators} \\
  \langle \quad \leq \quad \geq \quad \rangle \\
  \text{equal to, not equal to} \\
  \&\& \quad (a + b < \text{max}) \quad |\| \quad (\text{max} == 0 \quad \&\& \quad (a == b))
  \end{array} \]
Lecture 4. Functions and Modules

- **Functions** are the basic building blocks of C programs

- Programmer-defined functions
  - application-specific: good for only the application in which they appear
  - general-purpose: good for a wide range of applications

- Libraries hold collections of ‘standard’ general-purpose functions

| I/O         | Math     | Strings | Other     | ...
|-------------|----------|---------|-----------|-----
| printf      | sqrt     | strcmp  | rand      |     |
| fprintf     | sin      | strcpy  | malloc    |     |
| scanf       | cos      | strlen  | atoi      |     |
|             | ...      |         |           |     |

Use standard functions whenever possible; reuse, don’t reinvent

- A function **declaration** gives the types of the arguments and the return type

- A function **definition** is also a declaration plus a function **body**

- A function **body** is a compound statement that implements the function

- A function **call** invokes the named function, which executes, then returns

  - the **caller**, or calling function, is the function in which the function call appears
  - the **callee**, or called function, is the function that is invoked
Computing $e^x$

- Goal: write a program to approximate $e^x$, where $e = 2.718282…$

\[ e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots \]

where $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$

- Compute $e^x$ to a given precision: iterate until $e^x$ changes by less than the precision

For $x = 1.0$, precision = 0.0001

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x^i / i!$</th>
<th>$e^x$</th>
</tr>
</thead>
</table>
| 1   | 1.0000000 | 1.000000 | % lcc ex.c
| 2   | 1.000000  | 1.000000 % a.out |
| 3   | 0.500000  | 2.000000 |
| 4   | 0.166667  | 2.500000 |
| 5   | 0.041667  | 2.666667 |
| 6   | 0.008333  | 2.708333 |
| 7   | 0.001389  | 2.716667 |
| 8   | 0.000198  | 2.718056 |
| 9   | 0.000025  | 2.718254 |

Enter x and the precision: 1.00001
\[ e^{1.000000} = 2.718282; \text{ should be } 2.718282 \]

Enter x and the precision: 2.00001
\[ e^{2.000000} = 7.389047; \text{ should be } 7.389056 \]
Computing $e^x$, cont’d

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

float epowx(float x, float epsilon);

int main(void) {
    float precision, x, ex;

    printf("Enter x and the precision:\n");
    scanf("%f%f", &x, &precision);
    ex = epowx(x, precision);
    printf("e^%f = %f; should be %f\n", x, ex, exp(x));
    return 0;
}

float epowx(float x, float epsilon) {
    int i;
    float ex = 1.0, prevex = 0.0, num = 1.0, denom = 1.0;

    i = 1;
    while (fabs(ex - prevex) > epsilon) {
        prevex = ex;
        num *= x;
        denom *= i++;
        ex += num/denom;
    }
    return ex;
}
```
# Dissecting ex.c

```c
#include <math.h>

Includes the standard header `math.h`, which contains `declarations` for the standard library functions `exp` and `fabs`

```c
float epowx(float, float);
```

This `function declaration`, or `prototype`, says that `epowx` is a function that takes 2 `float` arguments and returns a `float` value

Functions must be declared (or defined) before they are used

```c
scanf("%f%f", &x, &precision);
```

`Calls` `scanf` to read two floating-point values ( `%f` ) and store them in `x` and `precision`

```c
ex = epowx(x, precision);
```

`Calls` `epowx` with the values of `x` and `precision` just read; `epowx` returns a `float`, which is stored in `ex`

```c
main is the `caller`, `epowx` is the `callee`
```

```c
printf("e^%f = %f; should be %f\n", x, ex, exp(x));
```

`Calls` `exp(x)` to compute the ‘real’ value of `e^x`, then `calls` `printf` with 4 arguments: a format string, the value of `x`, the value of `ex`, and the value returned by `exp`; `conversion specifier` `%f` prints the corresponding argument as a float
Dissecting ex.c, cont’d

float epowx(float x, float epsilon) {
  ...
}

The function definition for epowx; x and epsilon are the function parameters, both floats, and epowx returns a value of type float; { ... } contains the body

int i;
float ex = 1.0, prevex = 0.0, num = 1.0, denom = 1.0;

These declarations specify the local variables in epowx and initialize all but i

  i = 1;
while (fabs(ex - prevex) > epsilon) {
  prevex = ex;
  num *= x;
  denom *= i++;
  ex += num/denom;
}

This loop adds terms in the series until the difference between successive values of ex is less than or equal to epsilon; fabs is a standard library function

return ex;

This return statement returns the value of ex to the caller
Scope (a.k.a. Visibility)

- The *scope* of an identifier is that part of the program in which the identifier can be used.

- *Declarations* of parameters and local variables *introduce new identifiers*.
  
  The scope of a function parameter is the body of the function.
  
  The scope of a local variable extends from its declaration to the end of the compound statement in which the declaration appears.

- Identifiers in different scopes are *unrelated*, even if they have the same name.

```
int main(void) {
    float precision, x, ex;

    ...
    return 0;
}

float epowx(float x, float epsilon) {
    int i;
    float ex = 1.0, prevex = 0.0, ...;

    ...
    return ex;
}
```
Scope, cont’d

• Cannot declare the same identifier *twice* in the same scope

```c
float epowx(float x, float epsilon) {
    int x;
    ...  // error!
}
```

• Local declarations ‘hide’ parameter declarations and outer-level local declarations

```c
f(int x, int a) {
    int y, b;
    y = x + a*b;
    if (...) {
        int a, b;                           // a hides parameter a; b hides outer-level local b
        ...
        y = x + a*b;
    }
}
```

• Some consider it poor style to hide outer-level identifiers
Arguments and Locals

• **Local** variables are *temporary* variables
  
  *Created* upon entry to the function in which they are declared
  
  *Destroyed* upon return

• **Arguments** are transmitted *by value*
  
  the values of the actual arguments are *copied* to the formal parameters

• Arguments are *initialized local variables* and can be used just like any locals

  /* Illustrate call-by-value. */
  
  ```c
  #include <stdio.h>

  void f(int a, int x) {
    printf("a = %d, x = %d\n", a, x);
    a = 3;
    {
      int x = 4;
      printf("a = %d, x = %d\n", a, x);
    }
    printf("a = %d, x = %d\n", a, x);
    x = 5;
  }
  
  int main(void) {
    int a = 1, b = 2;
    f(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
  }
  
  % lcc args.c
  % a.out
  a = 1, b = 2
  a = 3, x = 2
  a = 3, x = 4
  a = 3, x = 2
  a = 1, b = 2
  ```

• Some consider it poor style to modify arguments
Global Variables

• A *global variable* is defined or declared outside of functions

• Globals are *permanent* variables
  
  *Created* when the program begins; *destroyed* when the program terminates

• The *scope* of global is from the point of declaration to the end of the file

  in file `foo.c`:

  ```c
  int main(void) {
      ...
  }
  int max = 0;
  void compute(...) {
      ...
  }
  ```

  • Parameters and locals ‘hide’ globals with the same names

    ```c
    void compute(...) {
        int max;  // local `max` hides global `max`
        ...
    }
    ```

  • Global variables *are* initialized to 0 by default
    (some consider it poor style to rely on this feature)
Modules

• A *module* is a set of related global variables and functions in one or more files

• *extern* *declarations* make globals and functions accessible from *other files*

  in file `baz.c`:

  ```c
  extern int max;
  void dump(...) {
      ...
  }
  ```

  The `max` defined in `foo.c` can be used here

• General-purpose *modules* are often packaged in *two* files

  The *interface* a header file (a `.h` file) of *declarations* for the variables and functions

  The *implementation* a `.c` file of *definitions* for those variables and functions

• Implementations can be *compiled separately*, and the compiled code can be stored in *libraries*
Modules, cont’d

random.h:

extern int random(void); /*
   returns a random number in the range 0..2147483646. */

extern int seed; /* Initial seed for random(); default 0. */

random.c:

/*
Random number generator; see Press et al.,
Numerical Recipes in C, 2/e, 278–9.
*/
#include "random.h"

int seed = 0;

int random(void) {
    int k;

    seed ^= 123459876;
    k = seed/127773;
    seed = 16807*(seed - k*127773) - 2836*k;
    if (seed < 0)
        seed += 2147483647;
    k = seed;
    seed ^= 123459876;
    return k;
}
Lecture 5. Arrays

• An array is a named collection of variables all of the same type
  Each variable in the collection is an element
  Elements are known by their integer positions or indices

  int count[11];
  defines an array named count that has 11 elements with indices 0..10

• Array elements are accessed by subscripting

  count[ expression ]

  expression is any expression whose value is an integer between 0 and 10 inclusive
  Subscript expressions are variables, a.k.a. lvalues

  No bounds checking — effect of out-of-bound subscripts is undefined

• Array elements occupy successive locations in memory

• Array elements are uninitialized; use loops to initialize them

  int i, count[11];

  for (i = 0; i < 11; i++)
    count[i] = 0;
Printing a Histogram

- **scores** contains 115 exam scores between 0 and 100

```c
/*
Print a histogram of scores from 0..100 in groups of 10.
*/
#include <stdio.h>

int main(void) {
    int i, counts[11], score;

    for (i = 0; i < 11; i++)
        counts[i] = 0;

    while (scanf("%d", &score) != EOF)
        counts[score/10]++;

    for (i = 10; i >= 0; i--) {
        int n = counts[i];
        printf("%3d ", 10*i);
        while (n-- > 0)
            printf("*");
        printf("\n");
    }

    return 0;
}
```

- Use an array to hold the number of scores in each 10-point range
Dissecting hist.c

```c
int i, counts[11], score;
for (i = 0; i < 11; i++)
  counts[i] = 0;

Declares counts and initializes each of its 11 elements to 0

while (scanf("%d", &score) != EOF)
  counts[score/10]++;

Reads the scores and increments the appropriate element of counts; scanf returns the value EOF at the end-of-file is reached (EOF is defined in stdio.h)

for (i = 10; i >= 0; i--) {
  int n = counts[i];
  printf("%3d ", 10*i);
  while (n-- > 0)
    printf("*");
  printf("\n");
}

Loops from counts[10] down to counts[0] printing each line of the histogram
```
Multidimensional Arrays

• Multidimensional arrays have two or more indices

```c
int x[3][5];
```
defines an 2-dimensional array `x` that has $3 \times 5 = 15$ elements

```
\begin{array}{cccccc}
  x[0][0] & x[0][1] & x[0][2] & x[0][3] & x[0][4] \\
  x[1][0] & x[1][1] & x[1][2] & x[1][3] & x[1][4] \\
\end{array}
```

• Array **rows** occupy successive locations in memory — *row-major order*

```
\begin{array}{cccccc}
  \_ & \_ & \_ & \_ & \_ \\
  x[0][0] & x[0][4] & x[1][0] & x[1][4] & x[2][0] & x[2][4] \\
\end{array}
```
Printing a Stem-and-Leaf Plot

• A stem-and-leaf plot displays the data values themselves in a histogram

```c
#include <stdio.h>

int main(void) {
    int i, j, counts[11][10], score;

    for (i = 0; i < 11; i++)
        for (j = 0; j < 10; j++)
            counts[i][j] = 0;

    while (scanf("%d", &score) != EOF)
        counts[score/10][score%10]++;

    for (i = 10; i >= 0; i--)
        for (j = 9; j >= 0; j--)
            printf("%d", j);

    printf("\n");

    return 0;
}
```

• Use a 2-dimensional array to hold the number of times each score occurs

Counts[i][j] is the number of times the score 10*i + j occurs

Each row of counts is a row in the stem plot
Dissecting stem.c

```c
int i, j, counts[11][10], score;
for (i = 0; i < 11; i++)
    for (j = 0; j < 10; j++)
        counts[i][j] = 0;

Declares counts as a 11-by-10 array and initializes each of its 110 elements to 0

counts[score/10][score%10]++;

Increments the element of counts that holds the number of times score occurs

for (i = 10; i >= 0; i--)
    printf("%2d ", i);
...
printf("\n");

Loops down the rows of counts, printing each ‘leaf’ and a new-line character

    for (j = 9; j >= 0; j--)
        int n = counts[i][j];
        while (n-- > 0)
            printf("%d", j);

Loops down the i-th column in counts printing j = score%10 for each occurrence of score```
Passing Arrays to Functions

- **Array parameters are declared by omitting the array size**

  ```c
  void record(int score, int counts[]) {
    counts[score/10]++;
  }
  ```

- **Arrays are passed to functions by giving just the array name**

  ```c
  void printhist(int n) {
    while (n-- > 0)
      printf("*");
  }
  ```

  ```c
  int main(void) {
    int i, counts[11], score;
    for (i = 0; i < 11; i++)
      counts[i] = 0;
    while (scanf("%d", &score) != EOF)
      record(score, counts);
    for (i = 10; i >= 0; i--)
      printf("%3d ", 10*i);
    printhist(counts[i]);
    printf("\n");
    return 0;
  }
  ```

- **Arrays — and only arrays — are passed in a way that simulates call-by-reference**
  
  The callee **can change** elements in the caller’s array argument
  
  An element is passed **by value** — the callee **cannot change** the caller’s element
Passing Arrays to Functions, cont’d

• Declare multidimensional array parameters by omitting only the number of rows

```c
void printstem(int counts[][10], int nrows) {
    while (--nrows >= 0) {
        int j;
        printf("%2d ", nrows);
        for (j = 9; j >= 0; j--) {
            int n = counts[nrows][j];
            while (n-- > 0)
                printf("%d", j);
        }
        printf("\n");
    }
}

int main(void) {
    int i, j, counts[11][10], score;
    for (i = 0; i < 11; i++)
        for (j = 0; j < 10; j++)
            counts[i][j] = 0;
    while (scanf("%d", &score) != EOF)
        counts[score/10][score%10]++;
    printstem(counts, 11);
    return 0;
}
```

• Passing the number of rows, or array size, to functions helps avoid indexing bugs
Lecture 6. Strings

• A **string** is an array of characters; quotes enclose **string constants**

```c
/* Everyone’s first
C program. */
#include <stdio.h>

int main(void) {
    'W', 'o', 'r', 'l', 'd', '!', '\0' };

    printf("%s\n", hello);
    return 0;
}
```

A string is terminated with a **null character** — the character with value 0

The **conversion specifier** %s causes the value of the corresponding string argument to be printed instead; i.e., its characters up to the null character

• Strings can be initialized with individual characters as above, or by

```c
char hello[] = "Hello World!";  // let the compiler count the characters
char *hello = "Hello World!";
```

char * declares a **character pointer**, which — for now — is the same as a string

• String variables can be used anywhere constant strings can be used

• Elements of string variables — the characters — can be changed by assignments
Printing Repeated Words

% lcc double.c
% echo Now is the the time / a.out
the
%

/* Print repeated words. */
#include <stdio.h>
#include <ctype.h>
#include <ctype.h>
#include <string.h>

int main(void) {
    char prev[100], word[100];

    prev[0] = '\0';
    while (scanf("%s", word) != EOF) {
        if (isalpha(word[0]) && strcmp(prev, word) == 0)
            printf("%s\n", word);
        strcpy(prev, word);
    }
    return 0;
}
Dissecting double.c

#include <ctype.h>
#include <string.h>

Includes the declarations for the character handling functions (ctype.h) and the string handling functions (string.h)

char prev[100], word[100];
prev[0] = '\0';

Declares two strings, prev and word, each capable of holding up to 100 characters, and initializes prev to the empty string

while (scanf("%s", word) != EOF) {
    ...
}

Loops reading the next string of nonblank characters into word

    if (isalpha(word[0]) && strcmp(prev, word) == 0)
        printf("%s\n", word);
    strcpy(prev, word);

Prints word if it begins with a letter (isalpha) and holds the same word as prev;

strcmp compares strings; then copies the string in word into prev (strcpy)

strcmp(x, y) returns a value <0, =0, >0 if x < y, x == y, x > y (lexicographic order)
Implementing String Handling Functions

• **strcpy(dst, src)** copies src to dst, character-by-character up to the ‘\0’
  
  ```c
  void strcpy(char dst[], char src[]) {
    int i;
    for (i = 0; src[i] != '\0'; i++)
      dst[i] = src[i];
    dst[i] = '\0';
  }
  ```

• **strcmp(str1, str2)** compares str1 and str2, character-by-character
  
  ```c
  int strcmp(char str1[], char str2[]) {
    int i;
    for (i = 0; str1[i] == str2[i] && str1[i] != '\0'; i++)
      ;
    if (str1[i] < str2[i])
      return -1;
    else if (str1[i] > str2[i])
      return +1;
    else
      return 0;
  }
  ```

• **Other string handling functions**

  - **strlen(str)** returns the number of nonnull characters in str
  - **strcat(dst, src)** appends src to the end of dst
Arrays of Strings

/* Shuffle a deck of cards. */
#include <stdio.h>
#include <stdlib.h>

char *suits[] = {
    "Hearts", "Diamonds", "Clubs", "Spades"
};

char *faces[] = {
    "Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"
};

int main(void) {
    int i, deck[52];
    deck[0] = 0;
    deck[1] = 1;
    for (i = 2; i < 52; i++) {
        int k = rand()%i;
        deck[i] = deck[k];
        deck[k] = i;
    }
    for (i = 0; i < 52; i++)
        printf("%s of %s\n", faces[deck[i]%13], suits[deck[i]/13]);
    return 0;
}
Dissecting shuffle.c

• Integer $k$ (0..51) represents the card with face value $k \% 13$ (0..12) and suit $k / 13$ (0..3)

```
char *suits[] = {
    "Hearts", "Diamonds", "Clubs", "Spades"
};
char *faces[] = {
    "Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"
};

Define and initialize global arrays of strings that map integers to suits and faces

deed[0] = 0;
deed[1] = 1;
for (i = 2; i < 52; i++) {
    int k = rand() % i;
    deck[i] = deck[k];
    deck[k] = i;
}

Initializes deck[0..51] to a random permutation of the integers 0..51

for (i = 0; i < 52; i++)
    printf("%s of %s\n", faces[deck[i] % 13], suits[deck[i] / 13]);

Prints the permuted deck in a readable form by mapping deck[i] % 13 (0..12) to a face and deck[i] / 13 (0..3) to a suit
Command-Line Arguments

• By convention, `main` is called with two arguments

```c
int main(int argc, char *argv[])
```

`argc` (‘argument count’) is the number of command-line arguments, including the program name.

`argv` (‘argument vector’) is an array of strings, one for each argument.

```bash
% echo Hello World
Hello World
%
```

• Implementing `echo`

```c
/* Echo my arguments. */
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    if (argc > 1)
        printf("%s", argv[1]);
    for (i = 2; i < argc; i++)
        printf(" %s", argv[i]);
    printf("
\n");
    return 0;
}
```

```bash
% lcc echo.c
% a.out Hello World
Hello World
%
```

Inside `main`:

```c
argc = 3
argv[0] = "a.out"
argv[1] = "Hello"
argv[2] = "World"
```
Testing random()

- **Check argc for optional command-line arguments**

```c
#include <stdio.h>
#include "random.h"

int main(int argc, char *argv[]) {
    int n = 100;
    if (argc > 1)
        sscanf(argv[1], "%d", &n);
    if (argc > 2)
        sscanf(argv[2], "%d", &seed);
    while (n-- > 0)
        printf("%d\n", random());
    return 0;
}
```

sscanf is like scanf, but reads from a string instead of from the input

% lcc testrandom.c random.c
% a.out / fmt
520932930 28925691 822784415 890459872 ... 100 random numbers
% a.out 1000 / fmt
520932930 28925691 822784415 890459872 ... 1000 random numbers
% a.out 4 126217318 / fmt
2088403071 1317687729 1526293439 721665858
Lecture 7. The TOY Machine

• TOY is an imaginary machine similar to early computers 1980s microprocessors

• Box with switches, lights, terminal

• TOY helps introduce machine-language programming (how a C program is ‘mapped’ onto a machine) computer architecture (how the machine works)

• With enough memory and time, TOY can compute anything a supercomputer can
Inside the Box

• 1 central processing unit (CPU)
• 256 16-bit **words** of memory
• 8 16-bit **registers**
• 1 8-bit **program counter** (PC) register — the address of the ‘current’ instruction
• Machine consists of ‘On/Off’ switches and lights

• Numbers are encoded in base 2, e.g., \( 6375_{10} = 0001\ 1000\ 1110\ 0111_2 \)

• **Operation**
  1. Load the program and the data into memory using the \( \text{addr} \), \( \text{data} \), and \( \text{load} \) switches
  2. Set the \( \text{addr} \) switches to the address of the first instruction
  3. Press \( \text{run} \)
  4. To examine memory — the ‘output’ — set \( \text{addr} \) switches to the desired address, press \( \text{look} \), read the \( \text{data} \) lights

• **Everything** is encoded in binary — data, machine instructions, text, addresses, …
Memory

- ‘Dump’ of machine state in hexadecimal includes the registers, PC, and memory
  
  \[
  \begin{align*}
  \text{PC} &= \text{000C} \\
  \text{R0} &= \text{0000} \quad \text{R1} &= \text{0037} \quad \text{R2} &= \text{0001} \quad \text{R3} &= \text{FFFF} \\
  \text{R4} &= \text{0000} \quad \text{R5} &= \text{0000} \quad \text{R6} &= \text{0008} \quad \text{R7} &= \text{00FF} \\
  \end{align*}
  \]

  00: 0000 0000 0000 0000 0000 0000 0000 0000
  08: 0000 0000 0000 0000 0000 0000 0000 0000
  10: 9222 9120 1121 A120 1121 A121 7211 0000
  18: 0000 0000 0000 0000 0000 0000 0000 0000
  20: 0000 0001 0010 0000 0000 0000 0000 0000
  28: 0000 0000 0000 0000 0000 0000 0000 0000
  ...
  E8: 0000 0000 0000 0000 0000 0000 0000 0000
  F0: 0000 0000 0000 0000 0000 0000 0000 0000
  F8: 0000 0000 0000 0000 0000 0000 0000 0000

- Programmers still look at dumps, even in the 90s

- Machine state
  
  records what a program has done
  determines what the machine will do
Basic Cycle

• When you press Run
  1. **Fetch**: load the instruction at the address given by the PC into the CPU
  2. **Increment** the PC by 1
  3. **Execute** the instruction, which may load/store data from/to memory
  4. Continue this *fetch-increment-execute cycle* until a halt is executed

• Instructions make well-defined changes to the registers, memory, and the PC
Digression: Number Systems

• The general form of an integer in base $b$ is

$$x = x_n b^n + x_{n-1} b^{n-1} + \ldots + x_1 b^1 + x_0 b^0$$

The $x_i$ are the positional coefficients

• Modern computers use binary arithmetic — base 2

$$140_{10} = 1 \times 10^2 + 4 \times 10^1 + 0 \times 10^0$$

$$= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= 10001100_2$$

• Base 2 is easily converted to base 8 (octal) and base 16 (hexadecimal)

$$140_{10} = 2 \times 8^2 + 1 \times 8^1 + 4 \times 8^0 = 214_8$$

$$= 8 \times 16^1 + C \times 16^0 = 8C_{16}$$

Digits in base 2 0 1
8 0 1 2 3 4 5 6 7
10 0 1 2 3 4 5 6 7 8 9
16 0 1 2 3 4 5 6 7 8 9 A=10 B=11 C=12 D=13 E=14 F=15
Conversions

• To convert from decimal to binary, divide by 2 repeatedly, read remainders up

\[
\begin{array}{c|c|c|c|c}
\hline
2 & 140 & 2 & 70 & 2 \times 0 \\
2 & 35 & 2 & 17 & 2 \times 0 \\
2 & 8 & 2 & 4 & 2 \times 1 \\
2 & 2 & 2 & 2 & 2 \times 0 \\
2 & 1 & 1 & 1 & 2 \times 1 \\
0 & 0 & 1 & 1 & 2 \times 0 \\
\hline
\end{array}
\]

Easier to convert to octal, then to binary, then to hexadecimal

\[
140 = \begin{array}{c|c|c|c|c|c|c}
& & \text{C} & \text{B} & \text{A} & \text{9} & \text{8} \\
& & 10001100 & 100 & 1100 & 100 & 0 \\
& & 2 & 1 & 4 \end{array}
\]

Easier to convert to octal, then to binary, then to hexadecimal

\[
140 = \begin{array}{c|c|c|c}
& & \text{C} & \text{B} \\
& & 1001100 & 100 \\
& & 2 & 1 \end{array}
\]
## Boolean Functions

- 16 possible Boolean functions of two binary variables; *some* have names, and C operators

<table>
<thead>
<tr>
<th>Truth table</th>
<th>Name</th>
<th>C expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>AND</td>
<td>x &amp; y</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>XOR</td>
<td>'exclusive or'</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>OR</td>
<td>'inclusive or'</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>NOR</td>
<td>'not or'</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>EQV</td>
<td>'not xor'</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>NOT y</td>
<td>one’s complement</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>XOR</td>
<td>'exclusive or'</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>OR</td>
<td>'inclusive or'</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>NOR</td>
<td>'not or'</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>EQV</td>
<td>'not xor'</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>NOT x</td>
<td>one’s complement</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>NOT</td>
<td>one’s complement</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>NAND</td>
<td>‘not and’</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>~</td>
<td></td>
</tr>
</tbody>
</table>

- Don’t confuse `&&` `||` `!` for `&` `|` `~`
Machine Arithmetic

• On a machine with 16-bit words, there are \(2^{16} = 65,536\) unsigned integers 0..65,535

\[
\begin{align*}
0000 \ 0000 \ 0000 \ 0000_2 & \quad 0 \\
0000 \ 0000 \ 0000 \ 0001 & \quad 1 \\
0000 \ 0000 \ 0000 \ 0010 & \quad 2 \\
0000 \ 0000 \ 0000 \ 0011 & \quad 3 \\
0000 \ 0000 \ 0000 \ 0100 & \quad 4 \\
\ldots \\
1111 \ 1111 \ 1111 \ 1100 & \quad 65,532_{10} \\
1111 \ 1111 \ 1111 \ 1101 & \quad 65,533 \\
1111 \ 1111 \ 1111 \ 1110 & \quad 65,534 \\
1111 \ 1111 \ 1111 \ 1111 & \quad 65,535
\end{align*}
\]

• There are 65,536 two's-complement signed integers \(-32,768..+32,767\)

\[
\begin{align*}
1000 \ 0000 \ 0000 \ 0000_2 & \quad -32,768_{10} \\
1000 \ 0000 \ 0000 \ 0001 & \quad -32,767 \\
\ldots \\
1111 \ 1111 \ 1111 \ 1110 & \quad -2 \\
1111 \ 1111 \ 1111 \ 1111 & \quad -1 \\
0000 \ 0000 \ 0000 \ 0000 & \quad 0 \\
0000 \ 0000 \ 0000 \ 0001 & \quad +1 \\
0000 \ 0000 \ 0000 \ 0010 & \quad +2 \\
\ldots \\
0111 \ 1111 \ 1111 \ 1110 & \quad +32,766 \\
0111 \ 1111 \ 1111 \ 1111 & \quad +32,767
\end{align*}
\]
Two’s-Complement Arithmetic

- Adding two’s-complement numbers is easy: Ignore signs, add unsigned numbers

<table>
<thead>
<tr>
<th></th>
<th>Start with</th>
<th>Complement</th>
<th>Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>+20</td>
<td>010100</td>
<td>111001</td>
<td>101100</td>
</tr>
<tr>
<td>+ - 7</td>
<td>+ 111001</td>
<td>+ + 7 + 000111</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>+13</td>
<td>001101</td>
<td>-13</td>
<td>110011</td>
</tr>
<tr>
<td>+20</td>
<td>010100</td>
<td>-20</td>
<td>101100</td>
</tr>
<tr>
<td>+ + 7</td>
<td>+ 000111</td>
<td>+ - 7 + 111001</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>+27</td>
<td>011011</td>
<td>-27</td>
<td>100101</td>
</tr>
</tbody>
</table>

- To negate a two’s complement number: Complement all the bits, then add 1
Lecture 8. TOY Instructions

- A program is a sequence of instructions
- An instruction is a 16-bit word, interpreted in one of many possible ways
- 3 instruction ‘formats,’ 16 different instructions

<table>
<thead>
<tr>
<th>Format 1</th>
<th>Format 2</th>
<th>Format 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0  halt</td>
<td>9  load</td>
<td>4  system call</td>
</tr>
<tr>
<td>1  add</td>
<td>5  jump</td>
<td></td>
</tr>
<tr>
<td>2  subtract</td>
<td>6  jump if less</td>
<td></td>
</tr>
<tr>
<td>3  multiply</td>
<td>7  jump indirect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8  jump and link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B  load immediate</td>
<td></td>
</tr>
</tbody>
</table>

Format 1: \[ \text{op} \quad \text{dst} \quad \text{reg}_1 \quad \text{reg}_2 \]
Format 2: \[ \text{op} \quad \text{dst} \quad \text{reg} \quad \text{con}_4 \]
Format 3: \[ \text{op} \quad \text{dst} \quad \text{con}_8 \]
Format 1 Instructions

<table>
<thead>
<tr>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>dst</td>
<td>reg₁</td>
<td>reg₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Format 1 instructions** are *register-to-register* instructions.
  - Interpret `dst`, `reg₁`, and `reg₂` as register numbers.
  - Take operands from `reg₁` and `reg₂`, and put the result in `dst`.
  - Example: \( \text{1234}_{16} \) means \( R_2 \leftarrow R_3 + R_4 \)

Stores the sum of the contents of registers \( R_3 \) and \( R_4 \) into register \( R_2 \)

- \( \text{2116}_{16} \) \( R_1 \leftarrow R_1 - R_6 \)
- \( \text{3267} \) \( R_2 \leftarrow R_6 \times R_7 \)
- \( \text{C512} \) \( R_5 \leftarrow R_1 \oplus R_2 \) exclusive OR
- \( \text{D645} \) \( R_6 \leftarrow R_4 \land R_5 \) logical AND
- \( \text{E056} \) \( R_0 \leftarrow R_5 > R_6 \) shift right
- \( \text{E764} \) \( R_7 \leftarrow R_6 < R_4 \) shift left
- \( \text{0000} \) halt
Format 2 Instructions

Format 2 instructions are memory operation instructions
Interpret \( dst \) and \( reg \) as register numbers, \( con4 \) as a 4-bit unsigned constant
Compute the effective address \( reg + con4 \)

- **Load** copies a word from memory at the effective address to register \( dst \)

\[ 9123_{16} \text{ means } R_1 \leftarrow M[R_2 + 3] \]

Copy the contents of the memory location specified by adding 3 to the contents of register \( R_2 \) to register \( R_1 \)

- **Store** copies a word from register \( dst \) to memory at the effective address

\[ A765_{16} \text{ means } M[R_6 + 5] \leftarrow R_7 \]

Copy the contents of register \( R_7 \) to the memory location specified by adding 5 to the contents of register \( R_6 \)

- When \( con4 \) is 0, load/store are sometimes called indirect load/store
Format 3 Instructions

<table>
<thead>
<tr>
<th>15</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>dst</td>
<td>con8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Most of the format 3 instructions are control instructions
  Interpret dst as a register number, con8 as an 8-bit unsigned constant or address
  Compute a jump address as either con8 or dst
  Set PC to that address
  Oddballs: system call (4) and load immediate (B)

• Load immediate copies con8 to register dst
  $B234_{16}$ means $R_2 \leftarrow 34_{16}$ set register R2 to $34_{16}$
  Use load immediate to copy the contents of a register to another register:
  $B000_{16}$ $R_0 \leftarrow 0$ set $R_0$ to 0
  $1320$ $R_3 \leftarrow R_2 + R_0$ set $R_3$ to $R_2 + R_0 = R_2 + 0 = R_2$

• System call invokes actions that need special permission, like I/O
  con8 specifies the system call ‘action code’, dst may specify an operand
  $A402_{16}$ writes the contents of $R_4$ to the standard output
Jump Instructions

- **Jump** instructions change the PC to \( con8 \), or to the contents of \( dst \)

\[
\text{jump} \\
5062_{16} \quad PC \leftarrow 62_{16}
\]

The next instruction will be taken from \( M[62_{16}] \)

\[
\text{jump if less} \\
6362 \quad PC \leftarrow 62_{16} \text{ if the contents of } R_3 < 0
\]

\[
\text{jump indirect} \\
7500 \quad PC \leftarrow R_5
\]

The next instruction will be taken from the address in \( R_5 \)

\[
\text{jump and link} \\
3A_{16} \quad 8462 \quad R_4 \leftarrow PC, \ PC \leftarrow 62_{16}
\]

\[
3B
\]

The contents of the PC \( (3B_{16}) \) are saved in \( R_4 \), then the PC is set to \( 62_{16} \)

The next instruction will be taken from \( M[62_{16}] \)

Used for **function linkage** — calls and returns

- All instructions of format 3 use a constant as one operand and a register or the program counter as the other operand.
Example: Bit Twiddling

- Set $b_0$ of $R_4$ to $b_{10} \land b_3$ from $R_1$, clear $b_1$–$b_{15}$ in $R_4$

$$R_4 = ((R_1>>10) \land (R_1>>3)) \land 1;$$

```
0101 0111 0111 0010  R1
0000 0000 0010 1001  R1>>10
0001 0100 1110 1110  R1>>3
0001 0100 1100 0111  (R1>>10) \land (R1>>3)
0000 0000 0000 0001  ((R1>>10) \land (R1>>3)) \land 1
```

Assuming $R_1$ is initialized to $A772_{16}$

```
00: B000 R0 <- 00
01: 1210 R2 <- R1 + R0 = A772
02: 1310 R3 <- R1 + R0 = A772
03: B50A R5 <- 0A
04: B603 R6 <- 03
05: E225 R2 <- R2 >> R5 = 0029
06: E336 R3 <- R3 >> R6 = 14EE
07: C323 R3 <- R2 \lor R3 = 14C7
08: B401 R4 <- 01
09: D443 R4 <- R4 \& R3 = 0001
```
Example: Polynomial Evaluation

- Evaluate \( ax^2 + bx + c = 2x^2 + 3x + 9 \) at \( x = 10 \) (239\(_{10} = \text{EF}_{16})

Store the ‘data’ in locations 30–33\(_{16}\)

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>000A</td>
</tr>
<tr>
<td>31</td>
<td>0002</td>
</tr>
<tr>
<td>32</td>
<td>0003</td>
</tr>
<tr>
<td>33</td>
<td>0009</td>
</tr>
</tbody>
</table>

- Use Horner’s method: rewrite \( ax^2 + bx + c \) as \( (ax + b)x + c \)

```assembly
10:  B330 R3 <- 30
11:  9430 R4 <- M[R3+00] = M[30] = 000A x
12:  9531 R5 <- M[R3+01] = M[31] = 0002 a
13:  3554 R5 <- R5 * R4 = 0014 a×x
15:  1556 R5 <- R5 + R6 = 0017 a×x + b
16:  3554 R5 <- R5 * R4 = 00E6 (a×x + b)×x
17:  9633 R6 <- M[R3+03] = M[33] = 0009 c
18:  1556 R5 <- R5 + R6 = 00EF (a×x + b)×x + c
19:  4502 system call 2: print R5 = 00EF
1A:  0000 HALT
```

- Polynomial evaluation for arbitrary \( x \)

  many applications, one \textit{raison d'etre} for early computers
Lecture 9. Branches and Loops

• Rewrite sum.c using labels and gotos

```c
#include <stdio.h>

int main(void) {
    int i = 1, n, sum = 0;

    printf("Enter n:\n");
    scanf("%d", &n);
    n--;
    Top: if (n < 0) goto End;
    sum += i;
    i++;
    n--;
    goto Top;
End: printf("Sum from 1 to %d = %d\n", i - 1, sum);
    return 0;
}
```

• Compilers implement C loop statements with branches and labels

while (conditional) 
statement 

L1: if (!conditional) goto L2
statement
      goto L1

L2:

Ditto for do-while and for loops
Implementing Loops, cont’d

0E
0E: B001  R0 <- 01  R0 holds 1
0F: B10A  R1 <- 0A  R1 is n
10: B201  R2 <- 01  R2 is i
11: B300  R3 <- 00  R3 is sum
12: 2110  R1 <- R1 - R0    n--
13: 6118  jump to 18 if R1 < 0  if (n < 0) goto End
14: 1332  R3 <- R3 + R2  sum += i
15: 1220  R2 <- R2 + R0  i++
16: 2110  R1 <- R1 - R0    n--
17: 5013  jump to 13  goto Top
18: 4302  print R3  print sum
19: 0000  halt

% /u/cs126/bin/toy /u/cs126/toy/sum.toy
Toy simulator $Revision: 1.8$
0037
PC = 001A
R0 = 0001  R1 = FFFF  R2 = 000B  R3 = 0037
R4 = 0000  R5 = 0000  R6 = 0000  R7 = 0000
0008: 0000 0000 0000 0000 0000 0000 0000 B001 B10A
0010: B201 B300 2110 6118 1332 1220 2110 5013
0018: 4302 0000 0000 0000 0000 0000 0000 0000
%
Example: Computing Fibonacci Numbers

Each number is sum of the previous two numbers; two numbers per loop iteration.

Computes Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... limit given by R2
Manipulating Addresses, a.k.a. Pointers

- Find the maximum value in an array of positive integers

```c
/*
   Find the largest positive integer
   in an array.
*/
#include <stdio.h>

short int a[15] = {
    0x0001, 0x0002, 0x0010, 0x1000, 0x7EFE,
    0x6030, 0x0040, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0010, 0x0000, 0x1000, 0x0000
};

int main(void) {
    int i = 14, max = 0;
    while (i >= 0) {
        if (a[i] > max)
            max = a[i];
        i--;
    }
    printf("%d\n", max);
    return 0;
}
```

% lcc max.c
% a.out
32510
Manipulating Addresses, cont’d

0E: B000 R0 <- 0 constant 0
0F: B101 R1 <- 1 constant 1
10: B20E R2 <- 0x0E i = 14
11: B322 R3 <- 0x22 address of a
12: B600 R6 <- 0 max = 0
13: 621B if R2 < 0 goto 1B while (i >= 0) {
14: 1723 R7 <- R2 + R3 R4 <- a[i]
15: 9470 R4 <- M[R7+0] = M[R2+R3]
16: 2546 R5 <- R4 - R6 compute a[i] - max
17: 6519 if R4-R6 < 0 jump to 19 if (a[i] > max)
18: 1640 R6 <- R4 + R0 = R4 max = a[i]
19: 2221 R2 <- R2 - R1 = R2 - 1 i--
1A: 5013 jump to 13 }
1B: 4602 print R6 print max
1C: 0000 halt

22: 0001 a[0]
23: 0002 28: 0040 2E: 0000
24: 0010 29: 0000 2F: 1000
25: 1000 2A: 0000 30: 0000 a[14]
26: 7EFE 2B: 0000
27: 6030 2C: 0000
2D: 0010

- R₂ + R₃ is the address of a[i]; R₂ is decremented, so R₂ + R₃ walks backwards in a
- What happens if location 11 is loaded with B318?
Function Linkages

- Use jump and link/jump indirect to call/return to/from functions

  \[
  \text{jump and link} \quad 8562 \quad R_5 \leftarrow \text{PC}, \text{PC} \leftarrow 62_{16} \\
  \text{jump indirect} \quad 7500 \quad \text{PC} \leftarrow R_5
  \]

- \textbf{power} computes \( R_3 \leftarrow x^n \) where \( x \) is \textit{passed} in \( R_1 \), \( n \) is \textit{passed} in \( R_2 \)

  \[
  \text{int power(int x, int n) \{ \\
  \quad \text{int } z = 1; \\
  \quad \text{while } (--n >= 0) \\
  \quad \quad z *= x; \\
  \quad \text{return } z; \\
  \}}
  \]

  
  
  14: B401 \quad R4 <- 1 \quad \text{constant 1} \\
  15: B301 \quad R3 <- 1 \quad z = 1 \\
  16: 2224 \quad R2 <- R2 - R4 = R2 - 1 \quad \text{while } (--n >= 0) \\
  17: 621A \quad \text{if } R2 < 0 \text{ jump to 1A} \\
  18: 3331 \quad R3 <- R3 * R1 \quad z *= x; \\
  19: 5016 \quad \text{jump to 16} \\
  1A: 7500 \quad \text{jump to address in R5} \quad \text{return } z
  

- \textit{Calling conventions} specify the locations of the actual arguments, the return value, and the return address; can vary among operating systems and languages on the \textit{same machine}.
Function Linkages, cont’d

• To compute $3^4 + 2^5$

```
04: B100   R0 <- 0
05: B11C   R1 <- 1C
06: 9110   R1 <- M[R1+0] = M[1C] = 0003
07: B204   R2 <- 4
08: 8514   call power, R5 <- 09
09: 1630   R6 <- R3 + R0 = R3 = 0051
0A: B11D   R1 <- 1D
0B: 9110   R1 <- M[R1+0] = M[1D] = 0002
0C: B205   R2 <- 5
0D: 8514   call power, R5 <- 0E
0E: 1663   R6 <- R6 + R3 = 0051 + 0020 = 0071
0F: 4602   print R6
10: 0000   halt
```

```
1C: 0003
1D: 0002

% /u/cs126/bin/toy /u/cs126/toy/power.toy
0071
PC = 0011
R0 = 0000   R1 = 0002   R2 = FFFF   R3 = 0020
R4 = 0001   R5 = 000E   R6 = 0071   R7 = 0000
```

• Function linkages on ‘real’ machines usually involve a stack to hold some of the arguments
Simulating TOY

- Any modern computer can **simulate** TOY: Write a C program that executes TOY instructions exactly as a TOY machine would

- Simulate memory and registers with 16-bit integer arrays

  ```c
  short int mem[256], regs[8];
  ```

- Simulate the PC and the **fetch-increment-execute** cycle

  ```c
  unsigned char pc;
  do {
      int inst = mem[pc++];
      execute(inst);
  } while (inst != HALT);
  ```

- **Switch statement** — a multiway branch — decodes and ‘executes’ instructions

  ```c
  void execute(int inst) {
      switch ((inst>>12)&0xF) {
      case ADD:
          regs[(inst>>8)&0F] = regs[(inst>>4)&0F] + regs[inst&0F];
          break;
      ...
      case JUMP: pc = inst&0xFF; break;
      }
  }
  ```

- This is simplified slightly; see `/u/cs126/toy/toy.c` for the full story
Lecture 10. Recursion

- A **recursive function** is a function that calls itself
  
  ```c
  int sum(int n) {
    if (n == 0)
      return 0;
    else
      return sum(n - 1) + n;
  }
  ```

- To compute \( f(n) \) using recursion
  
  - `compute f(0)`          `basis’ case
  - `compute f(n)` using \( f(k) \) for \( k < n \)   `recursive’ cases

- Recursion is like *mathematical induction*
  
  To prove \( S(n) \): prove \( S(0) \), then prove \( S(n) \) assuming \( S(k) \) for all \( k < n \)
  
  \[
  0 + 1 + 2 + 3 + \ldots + n = \frac{n(n + 1)}{2}
  \]

  Trivially true for \( n = 0 \)

  Assume it is true for \( 0 + \ldots + n - 1 \)

  Is it true for \( 0 + \ldots + (n - 1) + n \)?

  \[
  0 + 1 + 2 + \ldots + n = 0 + \ldots + (n - 1) + n = \frac{(n - 1)(n - 1 + 1)}{2} + n = \frac{n(n + 1)}{2}
  \]
Divide and Conquer

• Solve a problem by dividing it into smaller ones:
To compute \( \sqrt{n} \), find \( x \) such that \( n - x^2 = 0 \)

```c
float sqroot(float n, float l, float r) {
    float x = (l + r)/2.0;
    if (r - l < 0.000001)
        return x;
    else if (n - x*x < 0.0)
        return sqroot(n, l, x);
    else
        return sqroot(n, x, r);
}
```

```c
int main(int argc, char *argv[]) {
    int i;
    for (i = 1; i < argc; i++) {
        int n;
        sscanf(argv[i], "%d", &n);
        printf("sqrt(%d) = \%f (should be %f)\n", n, sqroot(n, 0.0, n), sqrt(n));
    }
    return 0;
}
```

% lcc sqroot.c
% a.out 5
sqrt(5) = 2.236068 (should be 2.236068)
**Binary Search**

- Suppose an array $x$ contains $n$ integers in increasing order; is $q$ in $x[0..n-1]$?

```c
int bsearch(int x[], int lb, int ub, int q) {
    if (lb <= ub) {
        int m = (lb + ub)/2;
        if (x[m] < q)
            return bsearch(x, m + 1, ub, q);
        else if (x[m] > q)
            return bsearch(x, lb, m - 1, q);
        else
            return m;
    } else
        return -1;
}
```

```c
k = bsearch(x, 0, 20 - 1, 26);
```
Number Conversion

- Print an integer in base $b$ (between 2 and 16)

```c
void convert(int n, int b) {
    if (n/b > 0)
        convert(n/b, b);
    printf("%c", "0123456789ABCDEF"[n%b]);
}
```

Printing 876 in base 5

\[
\begin{align*}
175 & \times 5 + 1 \\
(35 & \times 5 + 0) \times 5 + 1 \\
((7 & \times 5 + 0) \times 5 + 0) \times 5 + 1 \\
(((1 & \times 5 + 2) \times 5 + 0) \times 5 + 0) \times 5 + 1 \\
(((0 & \times 5 + 1) \times 5 + 2) \times 5 + 0) \times 5 + 0) \times 5 + 1 \\
1 & \times 5^4 + 2 \times 5^3 + 0 \times 5^2 + 0 \times 5^1 + 1 \times 5^0 \\
12001 & _5
\end{align*}
\]

In base 16

\[
\begin{align*}
54 & \times 16 + 12 \\
(3 & \times 16 + 6) \times 16 + 12 \\
((0 & \times 16 + 3) \times 16 + 6) \times 16 + 12 \\
3 & \times 16^2 + 6 \times 16^1 + 12 \times 16^0 \\
36C & _{16}
\end{align*}
\]
Pitfalls

• Many computations are expressed naturally as recursive functions
• **But**, some simple recursive functions consume excessive resources: compute $2^n$

```c
int f(int n) {
    if (n == 0)
        return 1;
    else
        return f(n-1) + f(n-1);
}
```

Hard way to compute $2^n$ because $f$ **recomputes intermediate results**

• Even ‘natural’ recursive function may consume excessive resources

```c
int fib(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return fib(n-2) + fib(n-1);
}
```

• Despite pitfalls, thinking and writing recursively yields correct implementations
• **Make it right before you make it fast**
Memo Functions

- Recursive functions can avoid recomputing intermediate results by saving them

```c
int fibs[51] = { 1, 1, 0 };  
int fib(int n) {  
  if (n <= 50 && fibs[n] != 0)  
    return fibs[n];  
  else {  
    int f;  
    if (n == 0 || n == 1)  
      f = 1;  
    else  
      f = fib(n-2) + fib(n-1);  
    if (n <= 50)  
      fibs[n] = f;  
    return f;  
  }
}
```

% lcc fib2.c
% a.out 30
fib(30) = 1346269

30: 1 14: 1597
29: 1 13: 2584
28: 2 12: 4181
27: 3 11: 6765
26: 5 10: 10946
25: 8  9: 17711
24: 13  8: 28657
23: 21  7: 46368
22: 34  6: 75025
21: 55  5: 121393
20: 89  4: 196418
19: 144  3: 317811
18: 233  2: 514229
17: 377  1: 832040
16: 610  0: 514229
15: 987
Changing Recursion to Iteration

- If the last action of a function is to call itself — ‘tail recursion’ — the call can be replaced with assignments and a loop; use labels and gotos, then a loop statement

```c
float sqroot(float n, float l, float r) {
    float x;
    x = (l + r)/2.0; // Loop: x = (l + r)/2.0;
    if (r - l < 0.000001)
        return x;
    else if (n - x*x < 0.0) // else if (n - x*x < 0.0)
        return sqroot(n, l, x); { r = x; goto Loop; }
    else // else
        return sqroot(n, x, r); { l = x; goto Loop; }
}

float sqroot(float n, float l, float r) {
    float x;
    x = (l + r)/2.0;
    while (r - l > 0.000001) {
        if (n - x*x < 0.0)
            r = x;
        else
            l = x;
        x = (l + r)/2.0;
    }
    return x;
}
```
Lecture 11. Quicksort

- Sort $x[0..n-1]$ into increasing (or decreasing) order
- Quicksort is a well-known sorting algorithm: Recursion is natural and fast

To sort $x[0..n-1]$:

1. Pick a ‘pivot’ element
2. Rearrange $x$ so that:
   - $x[k]$ holds this element, $x[0..k-1] < x[k]$, and $x[k+1..n-1] > x[k]$
3. Sort $x[0..k-1]$ and $x[k+1..n-1]$ recursively

```c
void quicksort(int x[], int l, int r) {
    if (r > l) {
        int k = partition(x, l, r);
        quicksort(x, l, k - 1);
        quicksort(x, k + 1, r);
    }
}

int main(void) {
    int n, array[1000];
    ...
    quicksort(array, 0, n - 1);
    ...
}
```
Partitioning

```c
int partition(int x[], int i, int j) {
    int k = j, v = x[k];
    i--;
    while (i < j) {
        while (x[++i] < v);
        while (--j > i && x[j] > v);
        if (i < j) {
            int t = x[i];
            x[i] = x[j];
            x[j] = t;
        }
    }
    x[k] = x[i];
    x[i] = v;
    return i;
}
```

Quick sort in Action

\[
\text{quick sort}(x, 0, 9) \quad 3 \ 8 \ 1 \ 7 \ 9 \ 0 \ 5 \ 2 \ 6 \ 4
\]
\[
3 \ 2 \ 1 \ 7 \ 9 \ 0 \ 5 \ 8 \ 6 \ 4
\]
\[
3 \ 2 \ 1 \ 0 \ 9 \ 7 \ 5 \ 8 \ 6 \ 4
\]
\[
3 \ 2 \ 1 \ 0 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
\text{quick sort}(x, 0, 3) \quad 3 \ 2 \ 1 \ 0 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
\text{quick sort}(x, 0, -1) \quad 0 \ 2 \ 1 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
\text{quick sort}(x, 1, 3) \quad 0 \ 2 \ 1 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
0 \ 2 \ 1 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
\text{quick sort}(x, 1, 2) \quad 0 \ 2 \ 1 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
\text{quick sort}(x, 1, 0) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
\text{quick sort}(x, 2, 2) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 7 \ 8 \ 6 \ 9
\]
\[
\text{quick sort}(x, 4, 3) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 7 \ 9
\]
\[
\text{quick sort}(x, 5, 9) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
\text{quick sort}(x, 5, 8) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 7 \ 5 \ 8 \ 6 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 7 \ 8 \ 6 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 7 \ 9
\]
\[
\text{quick sort}(x, 5, 5) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 7 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 7 \ 9
\]
\[
\text{quick sort}(x, 7, 8) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 7 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9
\]
\[
\text{quick sort}(x, 7, 6) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 7 \ 9
\]
\[
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 8 \ 9
\]
\[
\text{quick sort}(x, 8, 8) \quad 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 7 \ 9
\]
\[
\text{quick sort}(x, 10, 9)
\]
Implementing Recursive Functions

• Consider \( \text{sum}(10) \): each call must have its own argument \( n \) and its return address

• Use a stack to hold arguments, local variables, and the return address

```
sum(n=10) calls
   sum(9)
      sum(8)
         sum(7)
            sum(6)
               sum(5)
                  sum(4)
                     sum(3)
                        sum(2)
                           sum(1)
                              sum(0)
                                 returns 0
                                 returns 1
                                 returns 3
                                 returns 6
                                 returns 10
                                 returns 15
                                 returns 21
                                 returns 28
                                 returns 36
                                 returns 45
                                 returns 55
```
Implementing Recursive Functions, cont’d

• Use *conventions* for the stack and for how arguments, etc. are ‘pushed’

  Use \( R_7 \) as the ‘stack pointer;’ it holds the address of the top element

  Stack starts at FF\(_{16} \) and grows ‘down’ — toward *lower* addresses

  Push the arguments onto the stack before calling a function; push the return address upon entering a function

```
<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>B201</td>
<td>( \text{R2} \leftarrow 1 ) push the return address</td>
</tr>
<tr>
<td>31</td>
<td>2772</td>
<td>( \text{R7} \leftarrow \text{R7} - \text{R2} = \text{R7} - 1 )</td>
</tr>
<tr>
<td>32</td>
<td>A670</td>
<td>( \text{M}[\text{R7}+0] \leftarrow \text{R6} )</td>
</tr>
<tr>
<td>33</td>
<td>9171 W0</td>
<td>( \text{R1} \leftarrow \text{M}[\text{R7}+1] )</td>
</tr>
<tr>
<td>34</td>
<td>2312</td>
<td>( \text{R3} \leftarrow \text{R1} - \text{R2} = \text{R1} - 1 )</td>
</tr>
<tr>
<td>35</td>
<td>633D</td>
<td>Jump to ( \text{3D} ) if ( \text{R3} &lt; 0 ) if ( n = 0 ) return 0</td>
</tr>
<tr>
<td>36</td>
<td>2772</td>
<td>( \text{R7} \leftarrow \text{R7} - \text{R2} = \text{R7} - 1 ) push ( n - 1 )</td>
</tr>
<tr>
<td>37</td>
<td>A370</td>
<td>( \text{M}[\text{R7}+0] \leftarrow \text{R3} )</td>
</tr>
<tr>
<td>38</td>
<td>8630</td>
<td>( \text{R6} \leftarrow \text{PC} ), ( \text{PC} \leftarrow 30 ) call <em>sum</em></td>
</tr>
<tr>
<td>39</td>
<td>B201</td>
<td>( \text{R2} \leftarrow 1 ) pop ( n - 1 )</td>
</tr>
<tr>
<td>3A</td>
<td>1772 W0</td>
<td>( \text{R7} \leftarrow \text{R7} + \text{R2} = \text{R7} + 1 )</td>
</tr>
<tr>
<td>3B</td>
<td>9271</td>
<td>( \text{R2} \leftarrow \text{M}[\text{R7}+1] )</td>
</tr>
<tr>
<td>3C</td>
<td>1112</td>
<td>( \text{R1} \leftarrow \text{R1} + \text{R2} )</td>
</tr>
<tr>
<td>3D</td>
<td>9670</td>
<td>( \text{R6} \leftarrow \text{M}[\text{R7}+0] ) pop return address</td>
</tr>
<tr>
<td>3E</td>
<td>B201</td>
<td>( \text{R2} \leftarrow 1 )</td>
</tr>
<tr>
<td>3F</td>
<td>1772</td>
<td>( \text{R7} \leftarrow \text{R7} + \text{R2} = \text{R7} + 1 )</td>
</tr>
<tr>
<td>40</td>
<td>7600</td>
<td>( \text{PC} \leftarrow \text{R6} ) return</td>
</tr>
</tbody>
</table>
```
Implementing Recursive Functions, cont’d

• Main program makes the first call

<table>
<thead>
<tr>
<th>00: B000</th>
<th>R0 &lt;- 0</th>
<th>R0 holds 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: B7FF</td>
<td>R7 &lt;- FF</td>
<td>initialize stack pointer</td>
</tr>
<tr>
<td>02: B210</td>
<td>R2 &lt;- 50</td>
<td>R2 &lt;- address of n</td>
</tr>
<tr>
<td>03: 9220</td>
<td>R2 &lt;- M[R2+0]</td>
<td>R2 &lt;- n</td>
</tr>
<tr>
<td>04: B101</td>
<td>R1 &lt;- 1</td>
<td>push n</td>
</tr>
<tr>
<td>05: 2771</td>
<td>R7 &lt;- R7 - R1 = R7 - 1</td>
<td></td>
</tr>
<tr>
<td>06: A270</td>
<td>M[R7+0] &lt;- R2</td>
<td></td>
</tr>
<tr>
<td>07: 8630</td>
<td>R6 &lt;- PC, PC &lt;- 30</td>
<td>call sum</td>
</tr>
<tr>
<td>08: B201</td>
<td>R2 &lt;- 1</td>
<td>pop n</td>
</tr>
<tr>
<td>09: 1772</td>
<td>R7 &lt;- R7 + R2 = R7 + 1</td>
<td></td>
</tr>
</tbody>
</table>

0A: 4102 | print R1 | print sum(n) |
0B: 0000 | halt |
50: 0000 | n |
Lecture 12. Pointers

• Variables denote *locations in memory* that can hold values; arrays denote *contiguous locations*
  
  ```
  int i = 8, sum = -456;
  float average = 34.5;
  unsigned count[4];
  ```

• The *location* of a variable is its *lvalue* or *address*; the contents stored in that location is its *rvalue*

• A *pointer* is a variable whose *rvalue* is the *lvalue* of another variable — the address of that variable

• Pointers are typed: a ‘pointer to an int’ may hold only the lvalue of an int variable

  If `p` points to `sum`, `q` points to `count[2]`:

  ```
  int *p; unsigned *q;
  p = &sum;
  q = &count[2];
  ```

  `p` and `q` *cannot* point to `average`

• The *null pointer* — denoted NULL — points to *nothing*

  ```
  p = NULL;
  ```
Pointer Operations

- Two fundamental operations: creating pointers, accessing the values they point to
- Unary & ‘address of’ returns the address of its lvalue operand as an rvalue
- Unary * ‘indirection’ returns the lvalue given by its pointer operand’s rvalue

Suppose x and y are ints, p is a pointer to an int

\[ p = \&x; \quad p \text{ is assigned the address of } x \]
\[ y = *p; \quad y \text{ is the value pointed to by } p \]
\[ y = *(\&x); \quad \text{same as } y = x \]

- Declaration syntax for pointer types mimics the use of pointer variables in expressions

\[ \text{int } x, y; \]
\[ \text{int } *p; \quad *p \text{ is an int, so } p \text{ must be a pointer to an int} \]

- Unary * and & have higher precedence than most other operators

\[ y = *p + 1; \quad y = (*p) + 1; \]
\[ y = *p++; \quad y = *(p++); \]
Indirection

- Pointer indirection (e.g., *p) yields an **lvalue** — a **variable** — and pointer values can be manipulated like other values

  ```
  int x, y, *px, *py;
  px = &x; // px is the address of x
  *px = 0; // sets x to 0
  py = px; // py also points to x
  *py += 1; // increments x to 1
  y = (*px)++; // sets y to 1, x to 2
  ```

- Passing pointer arguments **simulates** passing arguments ‘by reference’

  ```
  void swap(int x, int y) {
    int t;
    t = x;
    x = y;
    y = t;
  }

  int a = 1, b = 2;
  swap(a, b);
  printf("%d %d\n", a, b);
  1 2
  ```
Pointers and Arrays

- Pointers can ‘walk along’ arrays by pointing to each element in turn

```c
int a[10], i, *p, x;
p = &a[0]; // p is assigned the address of the 1st element of a
x = *p;   // x is assigned a[0]
x = *(p + 1); // x is assigned a[1]
p = p + 1;  // p is assigned the address of a[1], by definition
p++;       // p points to a[2]
```

- Pointer arithmetic: If `p` points to `a[i]`, `p + k` points to `a[i+k]`

- An array name is a `constant` pointer to the first element of the array

```c
p = a;     // p is assigned the address of a[0]
a++;       // illegal: can’t change a constant
p++;       // legal: p is a variable
```

- The idiom `*p++` walks along the array pointed to by `p`

```c
p = a;
for (i = 0; i < 10; i++)
    printf("%d\n", *p++);
for (i = 0; i < 10; i++)
    printf("%d\n", a[i]);
```

Both loops print the same output, both are efficient, both are acceptable
Pointers and Array Parameters

• An array parameter type is identical to a pointer to the element type
  
  Array parameters are not constants, they are variables
  
  Passing an array as an actual argument passes a pointer to the first element
  
  In effect, arrays — and only arrays — are passed by-reference

```c
void print(int x[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d\n", x[i]);
}

void print(int *x, int size) {
    while (size-- > 0)
        printf("%d\n", *x++);
}
```

• A string is an array of characters; the name of a character array is thus a char *

• String functions can be written using arrays or pointers, but often return pointers

```c
char *strcpy(char *dst, char *src)
    copies src to dst, then returns dst

char *strcpy(char dst[], char src[]) {
    int i;
    for (i = 0; src[i] != '\0'; i++)
        dst[i] = src[i];
    dst[i] = '\0';
    return dst;
}
```
Pointers and Array Parameters, cont’d

• Pointer version

```c
char *strcpy(char *dst, char *src) {
    char *d = dst, *s = src;
    while (*d = *s) {
        d++;
        s++;
    }
    return dst;
}
```

• **Idiomatic** version

```c
char *strcpy(char *dst, char *src) {
    char *d = dst;
    while (*dst++ = *src++)
        while ((*dst++ = *src++) != ‘\0’);
    return d;
}
```

• Pointer versions *might* be faster, but strive for *clarity*, not microefficiency
Arrays of Pointers

• Arrays of pointers help build tabular structures

```c
char *suits[] = {
    "Hearts", "Diamonds", "Clubs", "Spades"
};

char *faces[] = {
    "Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"
};
```

Declare `suits` and `faces` each to be an ‘array of pointers to characters,’ *not* ‘a pointer to an array of characters’, and initialize them as shown

• Indirection (*) has *lower* precedence than []

```c
char *suits[];     // is the same as    char *(suits[]);
```

Declaration mimics use: `*suits[i]` refers to the 0th character in the `i`th string

```c
printsuit(int card) {
    printf("%c", *suits[card%13]);
}
```

• A string constant is shorthand for the name of an array of characters

```c
print("0123456789ABCDEF"[n%b]);    char digits[] = "0123456789ABCDEF大小写";
print(digits[n%b]);
```
Common Errors

- Only *addresses* can be assigned to pointers

```c
int *p, i;
p = i;  // Correct
p = &i; // Correct
```

- Only addresses of variables of the *correct types* can be assigned to pointers

```c
int *p;
float *p;
float x;
p = &x;
```

- Only pointers can be used with *indirection*

```c
p = *i;    // Correct
i = *p;    // Correct
```

- Pointers must be *initialized* to valid addresses *before* using indirection

```c
p = &i;
*p = 5;
printf("%d\n", *p);
```

- The null pointer must *not* be dereferenced, because it points to ‘nothing’

```c
p = NULL;
*p = 6;       // Error
```
Common Errors, cont’d

• Pointers must point to variables that exist! See page 4-8

```c
int *SumPtr(int a, int b) {
    int sum = a + b;
    return &sum;
}

p = SumPtr(2, 5);       \textit{sum does not exist!}
printf("%d\n", *p);
```

```c
char *itoa(int n) {
    char buf[100];
    sprintf(buf, "%d", n);
    return buf;
}
```

```c
char *s;
s = itoa(56);      \textit{buf does not exist!}
printf("%s\n", s);
```

`sprintf` is like `printf`, but stores the ‘output’ in a string

• When faced with bugs involving a pointer, ask: Is this pointer initialized? Does the memory it points to exist?
Lecture 13. Structures

• An array is a **homogeneous** collection: all of its elements have the same type

• A structure is a **heterogeneous collection**: its elements can have different types

```c
struct date {
    int day;
    int month;
    int year;
    char monthname[4]; /* "Jan", "Feb", etc. */
};
```

Declares a **new type**, `struct date`, with four named elements, called **fields**

• Structures can be **nested**

```c
struct student {
    char name[30];
    float gpa;
    struct date birthday;
};
```

• Structure types can be used like `int`, `float`, etc. to declare variables and arrays, which can optionally be initialized — and they must be initialized before use

```c
struct date today;
struct student cs126[140];
struct date bday = { 2, 11, 1977, "Nov" };```
Fields

• Structure fields are accessed by *variable.* *field*

  \[
  \begin{align*}
  \text{bday.day} & \quad \text{the day field in bday, the int 2} \\
  \text{bday.name[i]} & \quad \text{the ith character in the monthname field of bday, a char}
  \end{align*}
  \]

• *Field selection* operator associates to the *left* and has high precedence

\[
\begin{align*}
\text{struct student cs126[140];} \\
\text{cs126[i].gpa} & \quad \text{the GPA of the ith student in cs126} \\
\text{cs126[i].name[j]} & \quad \text{the jth character in the name of the ith student} \\
\text{cs126[i].birthday.year} & \quad \text{the year of the ith student’s birthday} \\
\text{cs126[i].birthday.monthname[0]} & \quad \text{the first letter in the monthname of the ith student’s birthday}
\end{align*}
\]

• Field selection denotes an **lvalue;** use assignments to initialize/change field values

\[
\begin{align*}
today.day & = 24; \\
today.month & = 10; \\
today.year & = 1996; \\
\text{strcpy(today.monthname, "Oct");}
\end{align*}
\]

\[
\text{swap(&today.day, &bday.day);} \\
\]

Arrays of Structures

• A structure type provides a way to package related data in one variable

```c
struct card {
    char *face;
    char *suit;
};

char *suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
char *faces[] = { Ace", "2", "3", "4", "5", "6", "7", "8",
                 "9", "10", "Jack", "Queen", "King" };

int main(void) {
    int i;
    struct card deck[52];
    deck[0].face = faces[0]; deck[0].suit = suits[0];
    deck[1].face = faces[1]; deck[1].suit = suits[0];
    for (i = 2; i < 52; i++) {
        int k = rand()%i;
        deck[i] = deck[k];
        deck[k].face = faces[i%13]; deck[k].suit = suits[i/13];
    }
    for (i = 0; i < 52; i++)
        printf("%s of %s\n", deck[i].face, deck[i].suit);
    return 0;
}
```

Once shuffled, cards are represented by `struct card` values, not integers 0..51
Pointers to Structures

• A **structure pointer** holds the address of a structure variable

```c
struct date today, bday, *pdate;

pdate = &today;                  // assigns the address of today to pdate
(*pdate).day = 2;                // sets the day field of today to 2
(*pdate).year++;                 // increments the year field of today
printf("%s %d, %d\n", (*pdate).monthname, (*pdate).day,
       (*pdate).year);            // prints the date given by today
bday = *pdate;                   // assigns today to bday, field-by-field
```

• Structure pointers can ‘walk along’ arrays of structures

```c
struct card *dptr;

dptr = deck;
for (i = 0; i < 52; i++) {
    printf("%s of %s\n", (*dptr).face, (*dptr).suit);
    dptr++;
}

dptr = dptr + 1;                 // increment dptr means
dptr +=1;                       // ‘advance dptr to the next struct card element’
dptr++;                         // *not* ‘add 1 to dptr’
```
Pointers to Structures, cont’d

• \((\texttt{*} \texttt{ptr}) \ . \texttt{field}\) is so common that there’s an abbreviation: \texttt{ptr->field}

  use \texttt{var} \ . \texttt{field} when \texttt{var} is a \texttt{structure}

  use \texttt{var->field} when \texttt{var} is a \texttt{pointer to a structure}

  or \texttt{(* var) . field}

-> has high precedence, but less than .

  \texttt{pdate->day = 2;} sets the \texttt{day} field of \texttt{*pdate} to 2

  \texttt{pdate->year++;} increments the \texttt{year} field of \texttt{*pdate}

  \texttt{printf("%s %d, %d\n", pdate->monthname, pdate->day, pdate->year);} prints the date given by \texttt{*pdate}

  \texttt{for (i = 0; i < 52; i++) { \}

        printf("%s of %s\n", dptr->face, dptr->suit);

        dptr++; \}

• Pointer madness! Structures can contain other pointers, but watch precedence

  \texttt{struct foo} \{ \texttt{int x, *y; } \} \texttt{*p;}

  \texttt{++p->x} increments field \texttt{x} in \texttt{*p}

  \texttt{(++p)->x} increments \texttt{p}, \texttt{then} accesses field \texttt{x}

  \texttt{*p->y++} returns the \texttt{int} pointed to by field \texttt{y} in \texttt{*p}, increments \texttt{y}

  \texttt{*p++->y} returns the \texttt{int} pointed to by field \texttt{y} in \texttt{*p}, increments \texttt{p}
Typedefs

- ‘struct card’ is a bit wordy and can make code hard to read
- A typedef associates an identifier with a type, which makes code more readable

```c
typedef struct card Card;
```

Declares Card to be a type name for ‘struct card’
Card may be used anywhere struct card can be used

Case matters!
Putting it all Together: Card Shuffling Revisited

- Represent a deck by an array of pointers to cards; shuffle by rearranging the pointers, not the cards themselves

```c
typedef struct card Card;
struct card {
    char *face;
    char *suit;
};
Card cards[52];
void shuffle(Card *deck[52]) {
    int i;
    deck[0] = &cards[0];
    deck[1] = &cards[1];
    for (i = 2; i < 52; i++) {
        int k = rand()%i;
        deck[i] = deck[k];
        deck[k] = &cards[i];
    }
}
```
Card Shuffling Revisited, cont’d

- Mapping of 0..51 onto faces and suits is confined to initialization

```c
char *suits[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
char *faces[] = { "Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King" };

void initialize(void) {
    int i;
    for (i = 0; i < 52; i++) {
        cards[i].face = faces[i%13];
        cards[i].suit = suits[i/13];
    }
}

int main(void) {
    int i;
    Card *deck[52];
    initialize();
    shuffle(deck);
    for (i = 0; i < 52; i++)
        printf("%s of %s\n", deck[i]->face, deck[i]->suit);
    return 0;
}
```

- Can handle *many* decks (arrays of pointers) with only *one* array of card structures
Lecture 14. Dynamic Memory Allocation

- The number of variables and their sizes are determined at **compile-time** — **before** a program runs

```c
#include <stdio.h>
#include "quicksort.h"

int main(void) {
    int i, n = 0, array[1000];

    while (n < 1000 && scanf("%d", &array[n]) == 1)
        n++;
    quicksort(array, 0, n - 1);
    for (i = 0; i < n; i++)
        printf("%d
", array[i]);
    return 0;
}
```

Suppose you want to sort 1001 integers? An unknown number of integers?
Size of the input is unknown at compile-time; it’s known only at runtime
Need a way for the program to **adapt** to the size of the input
Solution: **allocate** the array at runtime, not at compile time
Allocating Memory at Runtime

• To allocate 100 bytes of memory

```c
char *ptr;
ptr = malloc(100);
if (ptr == NULL) {
    printf("Cannot allocate memory\n");
    exit(1);
}
```

`malloc` allocates a contiguous block of memory at least 100 bytes long and returns the address of the first byte.

If `malloc` cannot allocate the memory requested, it returns `NULL` — *always* check! Better yet, use `emalloc` in `libmisc.a`

`malloc` returns a `generic pointer`, which can be assigned to any pointer type:

```c
strcpy(ptr, "Hello World!\n");
printf("%s", ptr);
```

Hello World!

• The memory block returned by `malloc` can be accessed *only* through a pointer; no variable labels that block
Deallocating Memory

• To deallocate the memory pointed to by "ptr"

  free(ptr);

  free deallocates the block of memory pointed to by "ptr"

  After calling free, "ptr" is \textit{uninitialized}; using this uninitialized value is an \textit{error}

• Memory blocks are allocated/deallocated by \textit{explicit calls} to malloc/free

  A block allocated by malloc exists until a call to free deallocates it

  malloc \textquotesingle creates\textquotesingle a block of memory, free \textquotesingle destroys\textquotesingle it

• The \textit{lifetime} of an allocated block is determined only by malloc/free; other function calls have \textit{no} effect on its existence

```c
char *itoa(int n) {
    char buf[100], *ptr;
    sprintf(buf, "%d", n);
    ptr = emalloc(strlen(buf) + 1);
    strcpy(ptr, buf);
    return ptr;
}
```

```c
char *s;

s = itoa(56);
printf("%s\n", s);
```

\textit{ptr} no longer exists, but the memory pointed to by \textit{s} \textit{does} exist!
## Sizeof

```c
int *SumPtr(int a, int b) {
    int *ptr, sum = a + b;
    ptr = emalloc(sizeof(int));  // how big is an int?
    *ptr = sum;
    return ptr;
}
```

```c
int *p = SumPtr(2, 5);
printf("%d\n", *p);
free(p);
```

- `sizeof (type)` is a **constant** that gives the size of values of *type* in bytes
  ```c
  ptr = emalloc(sizeof (int));  // allocate space for an int
  ```

- Values given by `sizeof` are machine-dependent

<table>
<thead>
<tr>
<th>Type</th>
<th>Sparc</th>
<th>Alpha</th>
<th>PCs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof (int)</code></td>
<td>4 bytes</td>
<td>4 or 8</td>
<td>2 or 4</td>
</tr>
<tr>
<td><code>sizeof (int *)</code></td>
<td>4</td>
<td>8</td>
<td>2, 4, or 8</td>
</tr>
<tr>
<td><code>sizeof (float)</code></td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof (int *)</code></td>
<td>4</td>
<td>8</td>
<td>2, 4, or 8</td>
</tr>
<tr>
<td><code>sizeof (void *)</code></td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

These values are only typical, not exhaustive
The size of a structure type may not be the sum of the sizes of its fields

```c
struct date {
    int day, month, year;
    char monthname[4];
};

struct student {
    char name[30];
    float gpa;
    struct date birthday;
};
```

```c
sizeof (struct date) 10–32 bytes
sizeof (struct student) 54–72
```

Use `sizeof` and `malloc/emalloc` to allocate instances of structure types

```c

struct date *mkdate(int day, int month, int year) {
    struct date *ptr = emalloc(sizeof (struct date));

    ptr->day = day; ptr->month = month; ptr->year = year;
    strcpy(ptr->monthname, months[month-1]);
    return ptr;
}
```
Dynamic Arrays

• To sort an arbitrary number of integers

Start with an array than can hold 1000 integers

*Double the size* of this array when more space is needed; 1000, 2000, 4000, ...

```c
#include <stdio.h>
#include "quicksort.h"
#include "misc.h"

int main(void) {
    int i, n = 0, *ptr, x, size = 1000;
    ptr = emalloc(size*sizeof (int));
    while (scanf("%d", &x) == 1) {
        if (n >= size) {
            size *= 2;
            ptr = erealloc(ptr, size*sizeof (int));
        }
        ptr[n++] = x;
    }
    quicksort(ptr, 0, n - 1);
    for (i = 0; i < n; i++)
        printf("%d\n", ptr[i]);
    return 0;
}
```
Dissecting sort2.c

#include "misc.h"

Includes the header file misc.h, which declares emalloc and erealloc

lcc -I/u/cs126/include sort2.c quicksort.c /u/cs126/lib/libmisc.a

Compiles sort2.c and quicksort.c, and searches libmisc.a to build a.out

int i, n = 0, *ptr, x, size = 1000;

Declares ptr and size (and i, n, and x), and initializes size to 1000

ptr = emalloc(size*sizeof (int));

Allocates space for size integers, and assigns the address of this array to ptr

while (scanf("%d", &x) == 1) {
        ...
        ptr[n++] = x;
    }

Reads each integer and assigns it to the next element in the array ptr

For any pointer ptr: ptr[i] is equivalent to *(ptr + i)

If ptr points to the first element of a dynamically allocated array:

    ptr + i points to the ith element,
    so ptr[i] refers to the ith element, too
Dissecting sort2.c, cont’d

if (n >= size) {
    size *= 2;
    ptr = realloc(ptr, size*sizeof (int));
}

Doubles the size of the array pointed to by ptr, if necessary

If n exceeds the current size of the array, size is doubled, and realloc is called to expand the array accordingly.

realloc returns the address of the expanded array, which is assigned to ptr.

realloc is like emalloc: It calls the standard library function realloc and checks for errors.

quicksort(ptr, 0, n - 1);
for (i = 0; i < n; i++)
    printf("%d\n", ptr[i]);

Sorts and prints the integers in ptr[0..n-1]
Common Errors

• Failing to allocate memory

    int *p, i;
    p = emalloc(sizeof (int));
    *p = i;

• Failing to allocate *enough* memory

    p = emalloc(sizeof (int *));
    *p = i;

    char *strsave(char *str) {
        return strcpy(emalloc(strlen(str)), str);
    }

• Deallocating memory that was *not* allocated by *malloc*

    char buf[100];
    free(buf);

• Deallocating memory that has *already been deallocated*

    p = emalloc(sizeof (int));
    free(p);
    ...
    free(p);
Common Errors, cont’d

• Changing the value of a pointer returned by `emalloc`, then passing it to `free`

```c
char *itoa(int n) {
    char buf[100];
    sprintf(buf, "%d", n);
    return strsave(buf);
}
```

```c
char *s = itoa(56); char *s = itoa(56), *p = s;
while (*s != '\0')
    putchar(*s++);
free(s);
free(p);
```

• Thinking that `sizeof` is a `runtime` operation

```c
int i, n, *p;

p = emalloc(sizeof (n)); p = emalloc(n*sizeof (int));
for (i = 0; i < n; i++)
    p[i] = 0;
```

• Failing to deallocate memory
Lecture 15. Dynamic Data Structures

- Pointers and structures can be used to build data structures that **expand** and **shrink** during execution, e.g., lists, stacks, queues, trees, ...

- Dynamic data structures are constructed using **self-referential** structure types

```c
struct node {
  int value;
  struct node *link;
};
```

 Declares a structure type with two fields

- `value` holds a integer
- `link` holds a pointer to a `struct node`

The type `struct node` is defined in terms of *itself* — self reference

```c
struct node n1, n2, n3;
```

- `n1.value = 4;`
- `n1.link = &n2;`
- `n2.value = 5;`
- `n2.link = &n3;`
- `n3.value = 6;`
- `n3.link = NULL;`

Builds a **singly linked list** with 3 nodes holding 4, 5, and 6
Lists

• Use a pointer to *traverse* a list — follow the link fields until you reach NULL

```c
struct node *p;
for (p = &n1; p != NULL; p = p->link) 4
    printf("%d\n", p->value); 5
6
```

• Use *emalloc/malloc* to allocate as many *struct* nodes as needed

```c
struct node *newnode = emalloc(sizeof (struct node));
newnode->value = 8;
newnode->link = NULL;
```

• To add a new node at the end of the list, walk a pointer down to the *last node*

```c
for (p = &n1; p->link != NULL; p = p->link) 4
    p->link = newnode;
```
List Headers

- Using a header node often simplifies list manipulations

```c
struct intlist {
    struct node *head;
    struct node *tail;
};
```

- Important boundary conditions

```c
struct intlist alist;
alist.head = alist.tail = NULL;
```

creates an *empty list*

so does

```c
struct intlist alist = { NULL, NULL };
```

```c
struct node *p = emalloc(sizeof (struct node));
p->value = 1;
p->link = NULL;
```

```c
alist.head = alist.tail = p;
```

creates a *one-node list*

- List headers can be allocated, too, if you need an arbitrary number of lists (as opposed to a list of arbitrary length)

```c
struct intlist *mylist = emalloc(sizeof (struct intlist));
```
A Simple List Module

• The *interface* defines the list types and list-manipulation functions

```c
/* Lists of ints */
struct intnode {
  int value;
  struct intnode *link;
};

struct intlist {
  struct intnode *head;
  struct intnode *tail;
};

extern void intlist_addhead (struct intlist *list, int value);
/* adds a new node holding value at the beginning of list */

extern void intlist_addtail (struct intlist *list, int value);
/* Adds a new node holding value at the end of list */

extern int intlist_remhead (struct intlist *list);
/* Removes the node at the beginning of a non-empty list
 and returns the value from that node */
```

This interface appears in *intlist.h*

• This kind of interface is an *abstract data type* because it defines a type and the operations on values of that type
Implementing the List Module

• The implementation defines the functions specified in the interface

/* Implementation of lists of ints */
#include <stdlib.h>
#include "intlist.h"
#include "misc.h"

void intlist_addhead(struct intlist *list, int value) {
    struct intnode *p = emalloc(sizeof (struct intnode));
    p->value = value;
    if (list->head == NULL) {
        p->link = NULL;
        list->head = list->tail = p;
    } else {
        p->link = list->head;
        list->head = p;
    }
}

extern int intlist_remhead(struct intlist *list) {
    ... }

void intlist_addtail(struct intlist *list, int value) {
    ... }

This implementation appears in intlist.c

• Adding a new node at the head of an intlist — beware boundary conditions

void intlist_addhead(struct intlist *list, int value) {
    struct intnode *p = emalloc(sizeof (struct intnode));
    p->value = value;
    if (list->head == NULL) {
        p->link = NULL;
        list->head = list->tail = p;
    } else {
        p->link = list->head;
        list->head = p;
    }
}
Implementing the List Module, cont’d

```c
void intlist_addtail(struct intlist *list, int value) {
    struct intnode *p = emalloc(sizeof (struct intnode));
    p->value = value;
    p->link = NULL;
    if (list->tail == NULL)
        list->head = list->tail = p;
    else {
        list->tail->link = p;
        list->tail = p;
    }
}
```

- When a node is deleted, it is also **deallocated**

```c
int intlist_remhead(struct intlist *list) {
    int value; struct intnode *p = list->head;
    if (list->head == list->tail)
        list->head = list->tail = NULL;
    else
        list->head = p->link;
    value = p->value; free(p);
    return value;
}
```

Wrong! Why?
Sorting Revisited

• Another way to sort an arbitrary number of integers

  1. Read them into an intlist, thus determining the number of integers
  2. Allocate an array
  3. Pour the integers in the list into the array
  4. Sort it and print it

#include <stdio.h>
#include "quicksort.h"
#include "intlist.h"
#include "misc.h"

int main(void) {
  int i, n, *ptr, x;
  struct intlist input = { NULL, NULL };
  for (n = 0; scanf("%d", &x) == 1; n++)
    intlist_addtail(&input, x);
  ptr = emalloc(n*sizeof (int));
  for (i = 0; i < n; i++)
    ptr[i] = intlist_remhead(&input);
  quicksort(ptr, 0, n - 1);
  for (i = 0; i < n; i++)
    printf("%d\n", ptr[i]);
  return 0;
}
Other Kinds of Lists

- **Stacks**: Add/remove nodes at only one end
  
  ```
  push  intlist_addhead
  pop   intlist_remhead
  ```

- **Queues**: Add nodes at the tail, remove nodes from the head
  
  ```
  put    intlist_addtail
  get    intlist_remhead
  ```

- **What about intlist_remtail?** Need a *doubly* linked list for efficient removal

- **Deques**: Add/remove nodes at either end
  
  ```
  push  intlist_addhead
  get   intlist_remhead
  put   intlist_addtail
  pull  intlist_remtail
  ```
Lecture 16. Writing Efficient Programs

• Is n a prime?

```c
int isprime(int n) {
    if (n > 2) {
        int i, m = n/2;
        for (i = 2; i < m; i++)
            if (n%i == 0)
                return 0;
        return 1;
    }
}

int main(int argc, char *argv[]) {
    int i;
    for (i = 1; i < argc; i++) {
        int n;
        sscanf(argv[i], "%d", &n);
        if (isprime(n))
            printf("%d is a prime\n", n);
        else
            printf("%d is not a prime\n", n);
    }
    return 0;
}
```

• 2147483647 is a prime, but isprime takes 1073741823 iterations to check!
Use a Better Algorithm

• Observations:

Need to check only odd integers

If \( n = a \times b \), then either \( a \) or \( b \) must be \( < \sqrt{n} + 1 \)

```c
#include <math.h>

int isprime(int n) {
    if (n > 2 && n%2 != 0) {
        int i, m = sqrt(n) + 1;
        for (i = 3; i < m; i += 2)
            if (n%i == 0)
                return 0;
    }
    return 1;
}
```

% lcc isprime2.c  
% a.out 2147483647  
2147483647 is a prime  

≈23169 iterations

• Better algorithms make programs faster, not microscopic code hacks

• Programs must be fast enough, not necessarily as fast as possible

• Don’t sacrifice clarity for speed
Searching

- A small ‘database’ problem: Maintain a list of names; lookup ‘queries,’ adding the new names, if necessary

```c
int main(int argc, char *argv[]) {
    int i;
    char buf[128];

    ptr = emalloc(size*sizeof (char *));
    ptr[0] = NULL;
    while (scanf("%s", buf) == 1)
        lookup(buf);
    for (i = 1; i < argc; i++) {
        int k = lookup(argv[i]);
        printf("%d\t%s\n", k, argv[i]);
    }
    printf("\n");
    for (i = 0; ptr[i] != NULL; i++)
        printf("%d\t%s\n", i, ptr[i]);
    return 0;
}
```

% lcc -I/u/cs126/include lookup.c /u/cs126/lib/libmisc.a
% a.out drh appel <names
3525 drh
794 appel
...
14210 zzwang
Searching, cont’d

• We know a good algorithm for searching — binary search (see page 10-3)

```c
int bsearch(char *x[], int lb, int ub, char *q) {
    if (lb <= ub) {
        int m = (lb + ub)/2;  
        int cond = strcmp(x[m], q);  
        if (cond < 0)  
            return bsearch(x, m + 1, ub, q);  
        else if (cond > 0)  
            return bsearch(x, lb, m - 1, q);  
        else  
            return m;
    } else  
    return -1;
}
```

• `ptr[0..count-1]` holds the names in ascending order; `ptr[count]` is NULL

```c
int count = 0;
char **ptr;

int lookup(char *name) {
    int k = bsearch(ptr, 0, count - 1, name);
    if (k == -1)  
        k = insert(strsave(name));
    return k;
}
```
Cost of Binary Search

• Counting *comparisons* — calls to `strcmp` in this version of `bsearch` — is a good measure of the cost of binary search

• Each recursive call cuts the problem in *half*, so the cost to search *N* names is

\[ C_N = C_{N/2} + 1 = C_{N/4} + 1 + 1 = \ldots \]

Suppose *N* = 2<sup>n</sup>, then

\[ C_{2^n} = C_{2^{n-1}} + 1 = C_{2^{n-2}} + 1 + 1 = \ldots = C_1 + 1 + \ldots + 1 = n \]

\[ C_N = \log_2 N = \lg N \]

Even for huge *N*, \( \lg N \) is small (conversely, even for small *n*, 2<sup>*n*</sup> is huge…)

<table>
<thead>
<tr>
<th><em>N</em></th>
<th>( \lg N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>(10^k)</td>
<td>(\approx 3.129 \times k)</td>
</tr>
</tbody>
</table>

• Bottom line: Binary search, and other \( \lg N \) algorithms, are *fast*
Inserting Names

- To keep the names in ascending order, `insert(q)`

  Expands the array, if necessary

  Slides `ptr[k..count-1]` down into `ptr[k+1..count]` where `ptr[k] > q`

  Stores `q` in `ptr[k]`, increments `count` and sets `ptr[count]` to `NULL`

  ```c
  int size = 1;
  int insert(char *q) {
      int k;
      if (count + 1 >= size) {
          size *= 2;
          ptr = erealloc(ptr, size*sizeof (char *));
      }
      for (k = count; k > 0 && strcmp(ptr[k-1], q) > 0; k--)
          ptr[k] = ptr[k-1];
      ptr[k] = q;
      ptr[+count] = NULL;
  }
  ```

- Oh oh… If the array holds `N` names, insert could take `N` comparisons
insert in Action

% echo P R I N C E T O N / a.out

the ‘hole’ moves over dimmed letters
## Binary Search Trees

- Different representations have different costs

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Insertion</th>
<th>Deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>fast</td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td>Linked list</td>
<td>slow: (\approx N)</td>
<td>fast w/search</td>
<td>fast w/search</td>
</tr>
<tr>
<td></td>
<td></td>
<td>slow w/o search</td>
<td>slow w/o search</td>
</tr>
<tr>
<td><strong>Binary tree</strong></td>
<td><strong>fast: (\approx \lg N)</strong></td>
<td><strong>fast</strong></td>
<td><strong>fast</strong></td>
</tr>
</tbody>
</table>

- In a binary search tree

```c
struct node {
    char *key;
    int info;
    struct node *left, *right;
};
```

Names in the **left** subtree are \(<\) than the name in the **root**

Names in **right** subtree are \(\geq\) the name in the root

Holds for any node in the tree
Searching in Binary Trees

- To search for $q$ in a binary search tree, start with $\text{tree} = \text{root}$
  
  1. If $\text{tree}$ is $\text{NULL}$, the search fails — an important boundary condition
  2. If $q < \text{tree->key}$, search the $\text{left}$ subtree
  3. If $q > \text{tree->key}$, search the $\text{right}$ subtree
  4. $q$ must be equal to $\text{tree->key}$

```c
struct node *search(struct node *tree, char *q) {
    if (tree != NULL) {
        int cond = strcmp(q, tree->key);
        if (cond < 0)
            return search(tree->left, q);
        else if (cond > 0)
            return search(tree->right, q);
        else
            return tree;
    } else
        return NULL;
}
```

- Cost of searching in $\text{balanced}$ binary trees is the same as for binary search in arrays — $\lg N$

- It’s possible to keep trees balanced during insertion; take COS 226, Data Structures, to find out how, and read R. Sedgewick, $\text{Algorithms in C}$, Addison-Wesley, 1990 (used in COS 226)
Searching, cont’d

int count = 0;
struct node *root = NULL;

int lookup(char *name) {
    struct node *p = search(root, name);
    if (p == NULL) {
        p = insert(root, NULL, strsave(name));
        p->info = count++;
    }
    return p->info;
}

int main(int argc, char *argv[]) {
    int i;
    char buf[128];
    while (scanf("%s", buf) == 1)
        lookup(buf);
    for (i = 1; i < argc; i++) {
        int k = lookup(argv[i]);
        printf("%d\t%s\n", k, argv[i]);
    }
    print(root);
    return 0;
}
Printing Trees

• Sorting is ‘free:’ Print the left subtree, print the key, print the right subtree

```c
void print(struct node *tree) {
    if (tree != NULL) {
        print(tree->left);
        printf("%d\t%s\n", tree->info, tree->key);
        print(tree->right);
    }
}
```

% lcc -I/u/cs126/include lookup2.c /u/cs126/lib/libmisc.a
% echo P R I N C E T O N | a.out

4 C
5 E
2 I
3 N
7 O
0 P
1 R
6 T

• Ways to traverse trees; ‘visit’ means ‘process the node,’ e.g., print its key

<table>
<thead>
<tr>
<th>preorder:</th>
<th>inorder:</th>
<th>postorder:</th>
</tr>
</thead>
<tbody>
<tr>
<td>visit</td>
<td>traverse left</td>
<td>traverse left</td>
</tr>
<tr>
<td>traverse left</td>
<td>visit</td>
<td>traverse right</td>
</tr>
<tr>
<td>traverse right</td>
<td>traverse right</td>
<td>visit</td>
</tr>
</tbody>
</table>
Inserting in Binary Trees

- insert is like search, but it must remember parent nodes in order to set the left or right field

```
A
  /   
G    H
 /     |
P      P
/      /  
M      N   
```

insert must also handle the empty tree, which occurs when parent is NULL
Inserting in Binary Trees, cont’d

```c
struct node *insert(struct node *tree, struct node *parent, char *q) {
    if (tree != NULL) {
        if (strcmp(q, tree->key) < 0)
            return insert(tree->left, tree, q);
        else
            return insert(tree->right, tree, q);
    } else {
        struct node *p = emalloc(sizeof (struct node));
        p->key = q;
        p->left = p->right = NULL;
        if (parent == NULL)
            root = p;
        else if (strcmp(q, parent->key) < 0)
            parent->left = p;
        else
            parent->right = p;
        return p;
    }
}

int lookup(char *name) {
    struct node *p = search(root, name);
    if (p == NULL) {
        p = insert(root, NULL, strsave(name));
        ...
    }
```
Lecture 17. Analysis of Algorithms

• An algorithm is a ‘method’ for solving a problem that is independent of a specific computer or programming language

• **Design**: Finding a way to solve the problem

• **Analysis**: Determining the algorithm’s cost in machine-independent terms, e.g. \( \lg N \)

• Need to make a program faster?

  Get a new machine

  Costs $$$ or more

  Makes ‘everything’ run faster

  But, it may — or **may not** — have much impact on a specific problem

  Get a new algorithm

  Costs ¢ or less

  Can make or break a specific problem by allowing it to be solved at all

  But, it may have **little or no** impact on ‘everything’
Sublist Sum Problem

• Given a list of numbers, find the contiguous sublist that has the largest sum

```
31  -41  59  26  -53  58  97  -93  -23  84
  187
```

```
31
31
31  -41  59
   49
31  -41  59  26
   75
31  -41  59  26  -53
   22
31  -41  59  26  -53  58
   80
31  -41  59  26  -53  58  97
  177
31  -41  59  26  -53  58  97  -93
   84
31  -41  59  26  -53  58  97  -93  -23
   61
31  -41  59  26  -53  58  97  -93  -23  84
  145
```

• Easy if all the numbers are nonnegative; tricky when some numbers are negative

• Sums must be positive; negative sublist sums are taken to be zero
A Simple Brute-Force Solution

• Try all possible sublists of \( n \) integers: \( x[\text{lb}..\text{ub}] \) for all \( \text{lb}, \text{ub} \) from 0 to \( n \)

```c
void sublist(int x[], int n) {
    int lb, ub, l, r, max = 0;
    for (lb = 0; lb < n; lb++)
        for (ub = lb; ub < n; ub++) {
            int i, sum = 0;
            for (i = lb; i <= ub; i++)
                sum += x[i];
            if (sum > max) {
                max = sum;
                l = lb;
                r = ub;
            }
        }
    printf("x[%d..%d] = %d\n", l, r, max);
}
```

% lcc -I/u/cs126/include sublistn3.c /u/cs126/lib/libmisc.a
% echo 31 -41 59 26 -53 58 97 -93 -23 84 | a.out
x[2..6] = 187
Profiling

• Program **profiles** help understand execution **frequencies**; use **lcc -b** and **bprint**

```bash
% lcc -b -I/u/cs126/include sublistn3.c /u/cs126/lib/libmisc.a
% echo 31 -41 59 26 -53 58 97 -93 -23 84 | a.out
x[2..6] = 187
% bprint
...
1 for (<1>lb = 0; <11>lb < n; <10>lb++)
2 for (<10>ub = lb; <65>ub < n; <55>ub++) {
    int i, sum = <55>0;
    3 for (<55>i = lb; <275>i <= ub; <220>i++)
        <220>sum += x[i];
    if (<55>sum > max) {
        <6>max = sum;
        <6>l = lb;
        <6>r = ub;
    }
}
1>printf("x[%d..%d] = %d\n", l, r, max);
```

• For **N = 10**

  Loop 1 is executed 11 \(\approx 10^1\) times
  2 65 \(\approx 10^{2/2}\)
  3 275 \(\approx 10^{3/3}\)

  Execution time \(\approx N^3\), can’t solve \(N = 10,000\), since \(10^{12}\) microseconds \(\approx 11\) days
A Better Algorithm

• Don’t recompute the whole sum every time

\[ x[lb] + x[lb+1] + \ldots + x[ub] = (x[lb] + \ldots + x[ub-1]) + x[ub] \]

```c
void sublist(int x[], int n) {
    int lb, ub, l, r, max = 0;
    for (lb = 0; lb < n; lb++) {
        int sum = 0;
        for (ub = lb; ub < n; ub++) {
            sum += x[ub];
            if (sum > max) {
                max = sum;
                l = lb;
                r = ub;
            }
        }
    }
    printf("x[%d..%d] = ", l, r);
}
```

```c
31 -41 59 26 -53 58 97 -93 -23 84
```

```c
printf("x[%d..%d] = ", l, r);
```

```c
31 -10 49 75 22 80 177 84 61 145
```

```c
printf("x[%d..%d] = ", l, r);
```

```c
59 85 32 90 187 94 71 155
```

```c
26 -27 31 128 35 12 96
```

```c
-53 5 102 9 -14 70
```

```c
58 155 62 39 123
```

```c
97 4 -19 65
```

```c
-93 -116 -32
```

```c
-23 61
```

```c
84
```
Profiling the Better Algorithm

for (<1>lb = 0; <11>lb < n; <10>lb++) {
    int sum = <10>0;
    for (<10>ub = lb; <65>ub < n; <55>ub++) {
        <55>sum += x[ub];
        if (<55>sum > max) {
            <6>max = sum;
            <6>l = lb;
            <6>r = ub;
        }
    }
}
<1>printf("x[%d..%d] = %d\n", l, r, max);

• For $N = 10$

  Loop 1 is executed $11 \approx 10^1$ times
  Loop 2 is executed $65 \approx 10^{2/2}$ times

  Execution time $\approx N^2$, but can’t solve $N = 1,000,000$, because $10^{12}$ microseconds $\approx 11$ days

• There is a divide-and-conquer algorithm that takes $\approx N \log N$, but there’s even a better way
The Optimal Algorithm

- Keep track of the maximum sum so far *and* the sum of the sublist that ends at \( x[i] \)

Suppose \( \text{max} \) is the maximum sum in \( x[0..i-1] \); extend that solution to \( x[i] \)

```c
void sublist(int x[], int n) {
    int i, l, r, max = 0, maxi = 0;
    for (i = 0; i < n; i++) {
        if (maxi + x[i] > 0)
            maxi += x[i];
        else
            maxi = 0;
        if (maxi > max)
            max = maxi;
    }
    printf("x[%d..%d] = %d
", l, r, max);
}
```

• Execution time \( \approx N \), because there’s just one loop; \( N = 1,000,000 \) takes \( \approx 1 \) second

• See `sublistn.c` for details of computing \( l \) and \( r \)
## Summary

- **A good algorithm can be more powerful than a supercomputer**

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
<th>Time (thousand)</th>
<th>Time (million)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force</td>
<td>$N^3$</td>
<td>17 min</td>
<td>300 centuries</td>
</tr>
<tr>
<td>Better</td>
<td>$N^2$</td>
<td>1 sec</td>
<td>11 days</td>
</tr>
<tr>
<td>Divide and Conquer</td>
<td>$N \log N$</td>
<td>0.01 sec</td>
<td>20 sec</td>
</tr>
<tr>
<td>Optimal</td>
<td>$N$</td>
<td>0.001 sec</td>
<td>1 sec</td>
</tr>
</tbody>
</table>

![Time Complexity Graph](image)

- For more, see J. Bentley, *Programming Pearls*, Addison Wesley, 1986
Lecture 18. Elementary Systems Programming

- **Software tools** are programs that **manipulate programs**, each in potentially different **languages**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro preprocessor</td>
<td>cpp</td>
<td>C</td>
</tr>
<tr>
<td>Compiler</td>
<td>rcc</td>
<td>assembly code</td>
</tr>
<tr>
<td>Assembler</td>
<td>as</td>
<td>object code</td>
</tr>
<tr>
<td>Linker</td>
<td>ld -r</td>
<td>object code, libraries</td>
</tr>
<tr>
<td>Loader</td>
<td>ld</td>
<td>object code</td>
</tr>
<tr>
<td>Operating system</td>
<td>UNIX</td>
<td>executable code</td>
</tr>
</tbody>
</table>

- ‘Driver’ programs, like lcc, hide many of these steps

```bash
% lcc -v hello.c
/usr/local/lcc/lib/cpp ... hello.c hello.i
/usr/local/lcc/lib/rcc -target=sparc-solaris hello.i hello.s
/bin/as -o hello.o hello.s
/bin/ld -o a.out ... hello.o -lm -lc
% a.out
Hello world!
```
Compilation Pipeline

% cat hello.c
/* Everyone’s first
     C program. */
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");
    return 0;
}

- The macro **preprocessor** strips comments, expands macro definitions, processes conditional compilation directives, and injects include files

% lcc -E hello.c >hello.i; cat hello.i
#line 1 "hello.c"
...
#line 1 "/usr/local/lib/lcc/include/stdio.h"
...
extern int printf(const char *, ...);
...
#line 4 "hello.c"

int main(void) {
    printf("Hello world!\n");
    return 0;
}

See Chap. 13 in Deitel and Deitel for details
Compilation and Assembly

• The *compiler* translates C to symbolic assembly language, alà symbolic TOY instructions

```c
% lcc -S hello.i; cat hello.s
...
_main:
save %sp, -96, %sp
set L2,%o0
call _printf; nop
mov %g0,%i0
L1:
ret; restore
L2: ...
```

• The *assembler* translates symbolic assembly language to relocatable object code, alà TOY instruction encodings

```c
% lcc -c hello.s; dis hello.o
0:  9d e3 bf a0  save   %sp, -96, %sp
4: 11 00 00 00  sethi  %hi(printf), %o0
8: 90 12 20 00  or     %o0, printf, %o0
c: 40 00 00 00  call    0xc
10: 01 00 00 00  nop
14: b0 10 00 00  clr     %i0
18: 81 c7 e0 08  ret
1c: 81 e8 00 00  restore
```
Linking

- **Linker** combines object code files and libraries in a new object code file

```c
% ld -r -o foo.o hello.o -lc; dis foo.o
```

```
main
0: 9d e3 bf a0 save %sp, -96, %sp
4: 11 00 00 00 sethi %hi(.L350), %o0
8: 90 12 20 00 or %o0, .L350, %o0
c: 40 00 00 00 call (.L350+12)
10: 01 00 00 00 nop
14: b0 10 00 00 clr %i0
18: 81 c7 e0 08 ret
1c: 81 e8 00 00 restore
```

```
printf
20: 9d e3 bf a0 save %sp, -96, %sp
24: 15 00 00 00 sethi %hi(0x0), %o2
28: f2 27 a0 48 st %i1, [%fp + 72]
2c: d4 02 a0 00 ld [%o2], %o2
30: 11 00 00 00 sethi %hi(0x0), %o0
34: f4 27 a0 4c st %i2, [%fp + 76]
```
Loading

• The **loader** translates object code to executable code

```%
ld foo.o; dis a.out
...
15148:  9d e3 bf a0  save  %sp, -96, %sp
1514c:  11 00 00 8a  sethi  %hi(0x22800), %o0
15150:  90 12 22 2c  or  %o0, 0x22c, %o0
15154:  40 00 00 05  call  0x15168
15158:  01 00 00 00  nop
1515c:  b0 10 00 00  clr  %i0
15160:  81 c7 e0 08  ret
15164:  81 e8 00 00  restore
15168:  9d e3 bf a0  save  %sp, -96, %sp
1516c:  13 00 00 e8  sethi  %hi(0x3a000), %o1
15170:  f2 27 a0 48  st  %i1, [%fp + 72]
15174:  d2 0a 62 b4  ldub  [%o1 + 692], %o1
15178: f4 27 a0 4c  st  %i2, [%fp + 76]
...
```

• The operating system loads the executable code into memory and jumps to it

```%
a.out
Hello world!
```
Assembly Language

• An *assembly language* is a *symbolic representation* for machine language
  Mnemonic names for opcodes and registers; usually terse
  Symbolic names for addresses — data locations and jump ‘targets’
  Easy to delete, insert, and rearrange instructions

• TAL:  _TOY Assembly Language_

<table>
<thead>
<tr>
<th>Command</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HALT</strong></td>
<td>halt</td>
</tr>
<tr>
<td><strong>ADD</strong></td>
<td>R,R₁,R₂ → R ← R₁ + R₂</td>
</tr>
<tr>
<td><strong>SUB</strong></td>
<td>R,R₁,R₂ → R ← R₁ − R₂</td>
</tr>
<tr>
<td><strong>MUL</strong></td>
<td>R,R₁,R₂ → R ← R₁ × R₂</td>
</tr>
<tr>
<td><strong>XOR</strong></td>
<td>R,R₁,R₂ → R ← R₁ ^ R₂</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>R,R₁,R₂ → R ← R₁ &amp; R₂</td>
</tr>
<tr>
<td><strong>SHL</strong></td>
<td>R,R₁,R₂ → R ← R₁ &lt;&lt; R₂</td>
</tr>
<tr>
<td><strong>SHR</strong></td>
<td>R,R₁,R₂ → R ← R₁ &gt;&gt; R₂</td>
</tr>
<tr>
<td><strong>LI</strong></td>
<td>R,const₈ → R ← const₈ (8-bit constant)</td>
</tr>
<tr>
<td><strong>LD</strong></td>
<td>R,(R₁ + const₈) → R ← M[R₁ + const₈]</td>
</tr>
<tr>
<td><strong>ST</strong></td>
<td>R,(R₁ + const₈) → M[R₁ + const₈] ← R</td>
</tr>
<tr>
<td><strong>SYS</strong></td>
<td>R,const₈ → system call system call const₈, arg in R</td>
</tr>
<tr>
<td><strong>J</strong></td>
<td>label → jump PC ← label</td>
</tr>
<tr>
<td><strong>JLT</strong></td>
<td>R,label → jump if less PC ← label if R &lt; 0</td>
</tr>
<tr>
<td><strong>JI</strong></td>
<td>(R) → jump indirect PC ← R</td>
</tr>
<tr>
<td><strong>JAL</strong></td>
<td>R,label → jump and link R ← PC, PC ← label</td>
</tr>
</tbody>
</table>
Programming in TAL

**power.t:** (see page 9-6)

```
POWER    LI R4,1  initialize R4
         LI R3,1  initialize z
LOOP     SUB R2,R2,R4  decrement exponent
         JLT R2, DONE  quit when done
         MUL R3,R3,R1  set z to z*x
         J LOOP  do it again
DONE     JI (R5)  return to caller
```

**main.t:** computes $A^4 + B^5$ (see page 9-7)

```
MAIN     LI R0,0  initialize R0
         LI R1, A  load A into R1
         LD R1, (R1+0)
         LI R2,4  want $A^4$
         JAL R5, POWER  call POWER
         ADD R6, R3, R0  copy $A^4$ to R6
         LI R1, B  do it again for $B^5$
         LD (R1+0)
         LI R2,5
         JAL R5, POWER
         ADD R6, R6, R3  R6 now holds $A^4 + B^5$
         SYS R6, 2  print it
         HALT
```

A

3

B

2
Object Code

• The assembler reads TAL and emits *relocatable object code*

  `power.o`:

  00: B401 =POWER initialize R4
  01: B301 initialize z
  02: 2224 decrement exponent
  03: 62 06 +start address quit when done
  04: 3331 set z to z\*x
  05: 50 02 +start address do it again
  06: 7500 return to caller

• Relocation information tells the *linker*

  The definitions of symbols

  How to adjust jump targets relative to the ultimate *starting address* of the module

  Which symbols are defined in other *separately compiled modules*

• Object code is usually a compact, binary format, not text as suggested above
Object Code, cont’d

main.o:

00:  B100 =MAIN initialize R0
01:  B100 +A load A into R1
02:  9110
03:  B204 want A\textsuperscript{4}
04:  8500 +POWER call POWER
05:  1630 copy A\textsuperscript{4} to R6
06:  B100 +B do it again for B\textsuperscript{5}
07:  9110
08:  B205
09:  8500 +POWER call POWER
0A:  1663 R6 now holds A\textsuperscript{4} + B\textsuperscript{5}
0B:  4602 print it
0C:  0000 =A
0D:  0003 =B
0E:  0002 =B

- Assemblers maintain symbol tables: Sets of (symbol,value) pairs used to map

  Mnemonics to values
  LI \rightarrow B_{16}, R6 \rightarrow 6, ...

  Labels to offsets
  LOOP \rightarrow 2_{16}, DONE \rightarrow 6_{16}, POWER \rightarrow 0, A \rightarrow 0D_{16}, ...

  power.o symbol table: (POWER, 0)

  main.o symbol table: (MAIN, 0) (A, 0D_{16}) (POWER, ?) (B, 0E_{16})

  Can implement symbol tables with binary trees
**Linking**

- The linker reads several object files and emits *one relocatable object code* file.

  *Concatenates* object code from input files

  *Relocates* symbol definitions and instructions based on starting addresses in output object code

- **Resolves** references to undefined symbols

- **Merges** symbol tables
Linking, cont’d

• Linking `main.o` and `power.o` resolves references to `POWER`, `A`, `B`, and adjusts offsets to jump targets

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>00:</strong></td>
<td>B100</td>
<td>=MAIN</td>
<td>initialize R0</td>
</tr>
<tr>
<td><strong>01:</strong></td>
<td>B10D</td>
<td>+start address</td>
<td>load A into R1</td>
</tr>
<tr>
<td><strong>02:</strong></td>
<td>9110</td>
<td></td>
<td>want A^4</td>
</tr>
<tr>
<td><strong>03:</strong></td>
<td>B204</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>04:</strong></td>
<td>850F</td>
<td>+start address</td>
<td>call POWER</td>
</tr>
<tr>
<td><strong>05:</strong></td>
<td>1630</td>
<td></td>
<td>copy A^4 to R6</td>
</tr>
<tr>
<td><strong>06:</strong></td>
<td>B10E</td>
<td>+start address</td>
<td>do it again for B^5</td>
</tr>
<tr>
<td><strong>07:</strong></td>
<td>9110</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>08:</strong></td>
<td>B205</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>09:</strong></td>
<td>850F</td>
<td>+start address</td>
<td>call POWER</td>
</tr>
<tr>
<td><strong>0A:</strong></td>
<td>1663</td>
<td></td>
<td>R6 now holds A^4 + B^5</td>
</tr>
<tr>
<td><strong>0B:</strong></td>
<td>4602</td>
<td></td>
<td>print it</td>
</tr>
<tr>
<td><strong>0C:</strong></td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>0D:</strong></td>
<td>0003</td>
<td>=A</td>
<td></td>
</tr>
<tr>
<td><strong>0E:</strong></td>
<td>0002</td>
<td>=B</td>
<td></td>
</tr>
<tr>
<td><strong>0F:</strong></td>
<td>B401</td>
<td>=POWER</td>
<td>initialize R4</td>
</tr>
<tr>
<td><strong>10:</strong></td>
<td>B301</td>
<td></td>
<td>initialize z</td>
</tr>
<tr>
<td><strong>11:</strong></td>
<td>2224</td>
<td></td>
<td>decrement exponent</td>
</tr>
<tr>
<td><strong>12:</strong></td>
<td>6215</td>
<td>+start address</td>
<td>quit when done</td>
</tr>
<tr>
<td><strong>13:</strong></td>
<td>3331</td>
<td></td>
<td>set z to z*x</td>
</tr>
<tr>
<td><strong>14:</strong></td>
<td>5011</td>
<td>+start address</td>
<td>do it again</td>
</tr>
<tr>
<td><strong>15:</strong></td>
<td>7500</td>
<td></td>
<td>return to caller</td>
</tr>
</tbody>
</table>

Output includes relocation information for additional linking
Loading

- The loader reads a *starting address* and object code with *no undefined symbols* and emits executable code, adding in the starting address where necessary.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>B100</td>
<td>initialize R0</td>
</tr>
<tr>
<td>21:</td>
<td>B1 2D</td>
<td>load A into R1</td>
</tr>
<tr>
<td>22:</td>
<td>9110</td>
<td>want A4</td>
</tr>
<tr>
<td>23:</td>
<td>B204</td>
<td>call POWER</td>
</tr>
<tr>
<td>24:</td>
<td>85 2F</td>
<td>copy A⁴ to R6</td>
</tr>
<tr>
<td>25:</td>
<td>1630</td>
<td>do it again for B⁵</td>
</tr>
<tr>
<td>26:</td>
<td>B1 2E</td>
<td>call POWER</td>
</tr>
<tr>
<td>27:</td>
<td>9110</td>
<td>R6 now holds A⁴ + B⁵</td>
</tr>
<tr>
<td>28:</td>
<td>B205</td>
<td>print it</td>
</tr>
<tr>
<td>29:</td>
<td>85 2F</td>
<td>do it again</td>
</tr>
<tr>
<td>2A:</td>
<td>1663</td>
<td>return to caller</td>
</tr>
<tr>
<td>2B:</td>
<td>4602</td>
<td>set z to z*x</td>
</tr>
<tr>
<td>2C:</td>
<td>0000</td>
<td>do it again</td>
</tr>
<tr>
<td>2D:</td>
<td>0003</td>
<td>return to caller</td>
</tr>
<tr>
<td>2E:</td>
<td>0002</td>
<td>return to caller</td>
</tr>
<tr>
<td>2F:</td>
<td>B401</td>
<td>initialize R4</td>
</tr>
<tr>
<td>30:</td>
<td>B301</td>
<td>initialize z</td>
</tr>
<tr>
<td>31:</td>
<td>2224</td>
<td>decrement exponent</td>
</tr>
<tr>
<td>32:</td>
<td>62 35</td>
<td>quit when done</td>
</tr>
<tr>
<td>33:</td>
<td>3331</td>
<td>set z to z*x</td>
</tr>
<tr>
<td>34:</td>
<td>50 31</td>
<td>do it again</td>
</tr>
<tr>
<td>35:</td>
<td>7500</td>
<td>return to caller</td>
</tr>
</tbody>
</table>
Separate Compilation

- A program is made up of many small modules
  - A ‘few’ application-specific modules
  - ‘Many’ general-purpose modules, e.g., standard I/O functions like printf
- Compile general-purpose modules separately, collect their object code in libraries
- Compile application-specific modules separately, keep their object code
- To build a program
  1. Link together the application-specific object code modules
  2. Search the libraries for the general-purpose modules used by (1)
- Advantages
  - Avoid recompiling infrequently changed modules
  - Share libraries of well-tested general-purpose modules — don’t reinvent, reuse
- Designing and implementing general-purpose modules sounds easy, but it’s not
  - Take COS 217, Introduction to Programming Systems
  - Read D. R. Hanson, C Interfaces and Implementations: Techniques for Creating Reusable Software, Addison-Wesley, 1997 (used in COS 217)
Lecture 19. Compilers

- The **compiler** translates a high-level language to a machine-level language

  \[ \text{lcc: } \quad \text{C} \rightarrow \text{SPARC assembly language} \rightarrow \ldots \rightarrow \text{SPARC machine code} \]

  \[ \text{compile: } \quad \text{arithmetic expressions} \rightarrow \text{TOY instructions} \]

- Most compilers have the basic phases

  - **Lexical Analysis** source code \( \rightarrow \) ‘tokens’
  - **Syntax Analysis** tokens \( \rightarrow \) abstract syntax trees
  - **Code Generation** abstract syntax trees \( \rightarrow \) machine-level code

- A compiler is a good example of

  - Application of theoretical computer science to a practical problem
  - Interaction between programming language design and computer architecture
  - Building a program from independent modules — ‘software engineering’

- For **much** more

  - Take COS 320, Compiler Design
  - Read C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, 1995
Lexical Analysis

• The lexical analyzer reads the source program and emits tokens or terminal symbols: the ‘letters’ in the ‘alphabet’ of the programming language

  English:
  
  a b c d e f g h ... A B C ... ; ‘ ’ ! : — - ( ) ...

  C tokens:
  
  if else while do for int float sizeof ...
  { } ; . -> + - * / % ++ -- < <= == != & ^ | ~ >= > ( ) ...

  "strings"  constants  identifiers ...

  Simple arithmetic expressions:

  ( ) + - *
  one-letter identifiers  one-digit constants

• A lexical analyzer usually discards white space: blanks, tabs, newlines, etc.

• Lexical analyzers can be described by and implemented with finite-state machines
Syntax Analysis

- A **context-free grammar** specifies how tokens can be formed into valid ‘sentences’

  Grammar rules or ‘productions’ specify how to generate all valid sentences

  1. \( \text{pgm} \rightarrow \text{expr} \)
  2. \( \text{expr} \rightarrow \text{expr} + \text{expr} \)
  3. \( \text{expr} \rightarrow \text{expr} - \text{expr} \)
  4. \( \text{expr} \rightarrow \text{expr} \ast \text{expr} \)
  5. \( \text{expr} \rightarrow ( \text{expr} ) \)
  6. \( \text{expr} \rightarrow \text{identifier} \)
  7. \( \text{expr} \rightarrow \text{constant} \)

  \( \text{pgm} \ \text{expr} \) are ‘nonterminals’ — they describe **classes** of valid sentences
  
  + − * ( ) identifier constant are terminals or tokens — the basic vocabulary

  1. \( \text{pgm} \Rightarrow \text{expr} \)
  3. \( \Rightarrow \text{expr} - \text{expr} \)
  5. \( \Rightarrow ( \text{expr} ) - \text{expr} \)
  4. \( \Rightarrow ( \text{expr} \ast \text{expr} ) - \text{expr} \)
  6. \( \Rightarrow ( \text{a} \ast \text{expr} ) - \text{expr} \)
  5. \( \Rightarrow ( \text{a} \ast ( \text{expr} ) ) - \text{expr} \)
  2. \( \Rightarrow ( \text{a} \ast ( \text{expr} + \text{expr} ) ) - \text{expr} \)
  6. \( \Rightarrow ( \text{a} \ast ( \text{b} + \text{expr} ) ) - \text{expr} \)
  7. \( \Rightarrow ( \text{a} \ast ( \text{b} + 2 ) ) - \text{expr} \)
  5. \( \Rightarrow ( \text{a} \ast ( \text{b} + 2 ) ) - ( \text{expr} ) \)
  2. \( \Rightarrow ( \text{a} \ast ( \text{b} + 2 ) ) - ( \text{expr} + \text{expr} ) \)
  6. \( \Rightarrow ( \text{a} \ast ( \text{b} + 2 ) ) - ( \text{c} + \text{expr} ) \)
  7. \( \Rightarrow ( \text{a} \ast ( \text{b} + 2 ) ) - ( \text{c} + 9 ) \)
Parsers

• A *parser* determines if a sentence can be generated by the grammar rules
  Proves that the sentence is syntactically valid

• A parser may also build an *abstract syntax tree* to represent the sentence
  \[(a \ast (b + 2)) - (c + 9)\]
  *Internal nodes* hold terminal symbols that denote operators: \(+ - \ast\)
  *Leaf nodes* hold terminal symbols that denote variables or constants: \(a b c 2 9\)

• A ‘recursive-descent’ parser has a function for each nonterminal
  ‘Matches’ terminals in input
  Calls other nonterminal functions — including itself — to apply the rules

• Parsers can be described by and implemented with *pushdown automata*
Code Generation

• A **code generator** traverses the abstract syntax tree and emits code, e.g., TAL, or TOY instructions

```c
% lcc -I/u/cs126/include compile.c /u/cs126/lib/libmisc.a
% a.out 5 6 7 "(a * (b + 2)) - (c + 9)"
```

```c
00: 0005
01: 0006
02: 0007
1A: 9100 R1 <- M[R0+0]
1B: 9201 R2 <- M[R0+1]
1C: B302 R3 <- 2
1D: 1223 R2 <- R2 + R3
1E: 3112 R1 <- R1 * R2
1F: 9202 R2 <- M[R0+2]
20: B309 R3 <- 9
21: 1223 R2 <- R2 + R3
22: 2112 R1 <- R1 - R2
23: 4102 print R1
24: 0000 halt
1A
```

• This compiler — and only this one — bypasses assembly, linking, and loading

```c
% a.out 5 6 7 "(a * (b + 2)) - (c + 9)" /u/cs126/toy/toy
ab2+*c9+-
Toy simulator $Revision: 1.14 $
0018
```
A Simple Compiler

• Lexical analyzer: returns characters as tokens
  
  ```c
  int get(char set[])  // returns the next token, advances the input
  int look(void)      // peeks at the next nonblank character
  ```

• Parser: returns an abstract syntax tree (AST)
  
  ```c
  Tree *expr(void)     // parses an expr, returns its AST
  Tree *pgm(char *string) // initializes lexer, parses a pgm, returns its AST
  
  struct tree {
    int op;
    struct tree *left, *right;
  };
  typedef struct tree Tree;
  
  Tree *maketree(int op, Tree *left, Tree *right) {
    Tree *t = emalloc(sizeof (Tree));
    t->op = op;
    t->left = left; t->right = right;
    return t;
  }
  ```

• Code generator: emits TOY instructions
  
  ```c
  int codegen(Tree *t, int dst, int loc)  // emits TOY code for AST t starting at loc
  ```
Lexical Analysis

• Globals hold the ‘state’ of lexical analysis: input and current input position

```c
char *input;  /* the "source code" */
int pos;  /* current position in input */

input[pos] holds the next character in the input
```

• The next token is the next non-whitespace character, which must be in set

```c
int get(char set[]) {
    while (isspace(input[pos]))
        pos++;
    if (input[pos] != '\0' && strchr(set, input[pos]) != NULL)
        return input[pos++];
    error("syntax error: expected one of '%s', set");
    return 0;
}
```

• The parser must peek ahead one character to determine its next action

```c
int look(void) {
    while (isspace(input[pos]))
        pos++;
    return input[pos];
}
```
Parsing

• The parsing functions for \textit{expr} and \textit{pgm} echo their grammar rules:

```c
Tree *expr(void) {
    Tree *t;
    if (look() == '(') { /* expr \rightarrow ( expr ) */
        get('('); t = expr(); get(')');
    } else if (isdigit(look())) /* expr \rightarrow constant */
        t = maketree(get("0123456789"), NULL, NULL);
    else /* expr \rightarrow identifier */
        t = maketree(get("abcdefghijklmnopqrstuvwxyz"), NULL, NULL);
    if (look() != '\0' && strchr("+-*", look()) != NULL) {
        int op = get("+-*"); /* expr \rightarrow expr [+-*] expr */
        t = maketree(op, t, expr());
    }
    return t;
}

Tree *pgm(char *string) {
    Tree *t;
    input = string; /* initialize lexical analyzer */
    pos = 0;
    t = expr(); /* pgm \rightarrow expr */
    if (look() != '\0')
        error("expected end of input\n");
    return t;
}
```
Reverse Polish Notation

• A postorder traversal of the AST yields a reverse Polish rendition of the expression

```c
void postorder(Tree *t) {
    if (t != NULL) {
        postorder(t->left);
        postorder(t->right);
        fprintf(stderr, "%c", t->op);
    }
}
```

(a * (b + 2)) - (c + 9)       a b 2 + * c 9 + -

• Reverse Polish can be evaluated: a stack holds operands and intermediate values

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>2</th>
<th>+</th>
<th>c</th>
<th>9</th>
<th>+</th>
<th>-</th>
<th>Stack→</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>5 6</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>5 6 2</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>5 8</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>40</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>40 7</td>
<td>40</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>40 7 9</td>
<td>40</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>40 16</td>
<td>40</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>2</td>
<td>+</td>
<td>c</td>
<td>9</td>
<td>+</td>
<td>-</td>
<td>24</td>
<td></td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>
```

• Instead of evaluating the expression, generate code, using registers for the stack
Code Generation

- **codegen** emits code to evaluate an AST into register `dst`, assuming higher numbered registers are free

```c
int codegen(Tree *t, int dst, int loc) {
    if (isalpha(t->op)) {
        int addr = t->op - 'a';
        printf("%02X: 9%X%X%X	R%d <- M[R%d+%d]\n", loc++,
            dst, 0, addr, dst, 0, addr);
    } else if (isdigit(t->op))
        printf("%02X: B%X%02X	R%d <- %d\n", loc++,
            dst, t->op - '0', dst, t->op - '0');
    else {
        loc = codegen(t->left, dst, loc);
        loc = codegen(t->right, dst + 1, loc);
        printf("%02X: %X%X%X\tR%d <- R%d %c R%d\n", loc++,
            strchr("+1-2*3", t->op)[1] - '0', dst,
            dst, dst + 1, dst, dst, t->op, dst + 1);
    }
    return loc;
}
```

Variables `a..z` are stored in locations `0..19_{16}`

`loc` is the location counter: the address of the next instruction emitted

`codegen` returns an updated value of `loc` for use by subsequent traversals
The Main Program

• The final touches

Arguments 1..argc–2 are the initial values of the corresponding variables

Argument argc–1 is the ‘source program’

Starting address is 26_{10} = 1A_{16}

```c
int main(int argc, char *argv[]) {
    Tree *e;
    int i, loc = 0;

    for (i = 1; i < argc - 1; i++)
        printf("%02X: %04X\n", loc++, atoi(argv[i]));
    if (i < argc) {
        e = pgm(argv[i]);
        postorder(e);
        fprintf(stderr, "\n");
        loc = codegen(e, 1, 26);
        printf("%02X: 4102 print R%d\n", loc++, 1);
        printf("%02X: 0000 halt\n", loc);
        printf("%02X\n", 26);
    }
    return 0;
}
```

See page 19-5 for an example of use
**Lecture 20. Operating Systems**

- An **operating system** provides a **virtual machine**: A high-level abstraction of an ugly low-level machine

- An OS provides **resources** and **services**
  - Memory management: Each user appears to have all the memory
  - Concurrency: Many users appear to compute simultaneously
  - Protection: User A can’t crash B’s program or access B’s files
  - File system: Files appear as streams of bytes, files have names, directories, random access
  - Interaction: X window system, window manager, mouse
  - Network access: The World Wide Web, remote file systems and printers

- **Programs communicate with the OS via system calls**, e.g. TOY opcode 4
  \[440_{16}\] prints the contents of \(R_4\)

  Each OS has its own (usually large) system call vocabulary
Multiprogramming

- A process is an executing instance of a program
  
  State includes registers, PC, memory management information

- The OS, a.k.a. kernel, multiplexes the processor between the processes, switching between processes at each interrupt

- When a periodic clock interrupt occurs (≈ every 1/60 second), do a context switch

  Stop
  Store the registers, PC, etc. in the current process’s state
  Load the registers, PC, etc. from the new process’s state
  Continue (‘dismiss the interrupt’)

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>emacs</td>
<td>lcc</td>
<td>a.out</td>
</tr>
<tr>
<td>emacs</td>
<td>lcc</td>
<td>a.out</td>
</tr>
<tr>
<td>emacs</td>
<td>lcc</td>
<td>a.out</td>
</tr>
</tbody>
</table>
Reentrant Programs

- A *reentrant program* does not modify its own code; it changes only its data.
- One copy can be shared among many processes; each process has its own data.

**Three processes running `emacs`**

Reentrant programs use less memory.

- What about the *addresses* in each process?
Virtual Memory

• Problem 1
  Several programs need to use the same memory
  Direct solution: Divide up the memory

• Problem 2
  If the OS can load program anywhere in memory, what is its starting address?
  Direct solutions: Have OS adjust relocatable addresses upon loading
  Use only position-independent code (impossible in TOY)

• Problem 3
  One program needs more memory than the machine has, or more than is left
  Direct solution: ‘Overlay’ unused functions with other functions

• ‘Better’ solution to all these problems
  Each program assumes access to the entire memory — its virtual address space
  Hardware helps OS associate a small part of physical address space with each process, keep some of the virtual address space on disk
Paging

- **Paging** is the predominant method for implementing virtual memory
- Maximum *effective address* determines the virtual address space size
- Divide physical memory and virtual memory into fixed-size ‘pages’
  - Use a power of 2
  - Leading address bits give the page number
  - Trailing address bits give the offset in that page
  - Example: 16-bit addresses, 8-bit page #s, 256-byte pages
- Build hardware to map all addresses through a *page table*
  - Indexed by virtual page #
  - Maps virtual page # → physical page #
  - Indicates whether page is in memory or on disk
  - Indicates whether in memory page is ‘dirty’ or clean
- Keep virtual memory for each program on disk
Each page read in from disk has to replace another page: Use page replacement strategies, such as Least Recently Used.
Size of Virtual Memory

• 16 bits is not enough
• 24 bits is not enough
• 32 bits is not enough!
• Is 64 bits enough?
  
  18,446,744,073,709,551,616 > 10^{19} \text{ addresses}

• 64-bit address space needs more sophisticated paging strategy and hardware
  Page table would be too big: 2^{13} = 8\text{Kbyte pages needs } 2^{51} \text{ page-table entries}
  Associative page tables, multilevel page tables

• Some big numbers

  \begin{align*}
  10^{20} & \quad \text{Number of grains of sand on a beach} \\
  10^{27} & \quad \text{Number of oxygen atoms in a thimble} \\
  2^{256} > 10^{77} & \quad \text{Number of electrons in the universe}
  \end{align*}
File Systems

• Disks are messy: Rotating cylinders with movable heads
  Rotational latency: Wait for the ‘track’ to appear under the head
  Seek time: Wait for the head to move in/out to the cylinder
  At best, a disk is an array of fixed-size blocks

• A file system provides high-level features on low-level disks
  Directories
  Named files
  Read/write arbitrary number of bytes
  Random access
  Automatic growth
UNIX File System

- Disk, array of fixed size blocks, is divided into 3 regions
  - Root block: File system parameters \( M, N \), list of free data blocks
  - ‘Inode’ blocks: Hold ‘information’ nodes, one per file or directory
  - Data blocks: Hold the data, file names in directories
- Inode blocks each hold \( k \) inodes numbered 0 to \( k-1 \), so a file system can hold \( k \times M \) files/directories
- An inode holds everything about a file, except its name
  - Type: directory or file
  - Size in bytes
  - Block numbers of its data blocks or indirect blocks
  - Number of directories pointing to the file
  - Times of creation, last modification
- A directory is just list of (file name, inode number) pairs
File Layout

- **Small file**: Inode points to 10 data blocks
  
  For 1Kbyte data blocks, handles files ≤ 10 Kbyte

- **Medium-size file**: Inode points to 10 ‘indirect’ blocks that point to data blocks
  
  With 4-byte block #s, handles files ≤ 10×256×1024 = 2,621,440 = 2.5 Mbyte

- **Large files**: Entries in last indirect block point to other indirect blocks
  
  Handles files ≤ (9 + 256)×256×1024 = 69,468,160 = 66.25 Mbyte

- **Huge files**: Inode points to 10 indirect blocks that each point to 256 indirect blocks
  
  Handles files ≤ 10×256×256×1024 = 671,088,640 = 640 Mbyte

- **Adjust block size/inode size to span larger disks**
Typical Medium-Size File
Lecture 21. Regular Expressions

• A regular expression describes a set of strings by giving a ‘pattern’ for them
  
  \(c\) Any nonspecial character matches itself \(A\)
  
  Any single character \(x\)
  
  Special character \(c\) \(\backslash\). \([\ldots]\) Any character in ..., including ranges \([a-z0-9]\)
  
  \([\ldots]\) Any character not in ..., including ranges \([^{0-9}]\)
  
  Whatever matches \(R_1\) followed by \(R_2\) \([A-Z]_\) \(R^*\) Zero or more occurrences of \(R\) \([a-z][a-z]*\)

• Tokens in most programming languages can be described by regular expressions
  
  \([1-9][0-9]*\) Decimal constants in C
  
  \([0-7]*\) Octal constants in C
  
  \([0-9][0-9]*\backslash . [0-9]*\) Floating constants in C
  
  \([A-Za-z_][A-Za-z_0-9]*\) C identifiers
  
  "[^["\n]*" String literals in C
  
  ’"[^["\n]*’ (quoted for the shell)
**egrep**

- Many UNIX tools support searching for patterns described by regular expressions
  - `egrep`, `grep`, `fgrep` - Search for lines matching regular expressions
  - `ed`, `vi`, `emacs` - Text editors
  - `sed` - Stream editor
  - `awk` - String-processing language
  
  More ...

- `egrep` prints those lines that match the regular expression

```bash
% cd /u/cs126/examples
% egrep emalloc *.c
compile.c:   Tree *t = emalloc(sizeof (Tree));
intlist.c:   struct intnode *p = emalloc(sizeof (struct intnode));
intlist.c:   struct intnode *p = emalloc(sizeof (struct intnode));
lookup.c:    ptr = emalloc(sizeof (char *));
lookup2.c:   struct node *p = emalloc(sizeof (struct node));
sort2.c:     ptr = emalloc(sizeof (int));
sort3.c:     ptr = emalloc(sizeof (int));
sublistn.c:  array = emalloc(sizeof (int));
sublistn2.c: array = emalloc(sizeof (int));
sublistn3.c: array = emalloc(sizeof (int));
```
egrep, cont’d

• /usr/dict/words contains \approx 25,143 words

  % egrep hh /usr/dict/words
  beachhead
  highhanded
  withheld
  withhold

  How many words have 3 a’s one letter apart?

  % egrep .a.a.a /usr/dict/words | wc -l
  50
  % egrep .u.u.u /usr/dict/words
  cumulus

• egrep supports extended regular expressions

  ^                 Beginning of line
  $                 End of line
  \( R^+ \)         One or more occurrences of \( R \)        \[0−9]^+
  \( R? \)         Zero on one occurrence of \( R \)        \[0−9]^{*\ .\ ?[0−9]^+}
  \( R_1|R_2 \)   Whatever matches \( R_1 \) or \( R_2 \)        \[A−Z]\_^{+}
  ( \( R \) )      Grouping
egrep, cont’d

• egrep as a simple spelling checker: Specify plausible alternatives you know

  % egrep "n(ie|ei)ther" /usr/dict/words
  neither

• Find big files; du –ka prints file sizes in 1Kbyte blocks

  % du –ka /etc | egrep ’^[5-9][0-9][0-9]’
  500 and up
  552  /etc/fs/nfs/mount
  553  /etc/fs/nfs
  837  /etc/fs
  850  /etc/lp/printers
  883  /etc/lp

• Find all lines with signed numbers

  % egrep ’[+-][0-9]+\.[0-9]*’ *.c
  bsearch.c: return -1;
  compile.c:                  strchr("+1-2*3", t->op)[1] - ’0’, dst,
  convert.c:Print integers in a given base 2-16 (default 10)
  convert.c: sscanf(argv[i+1], "%d", &base);
  ...
  strcmp.c: return -1;
  strcmp.c: return +1;

• egrep has its limits: It cannot match all lines that contain a number divisible by 5
Formal Languages

• A **language** is a (possibly infinite) set of strings over a finite alphabet

• A regular expression describes a language: The set of all strings it ‘matches’

• A **regular language** is any language that can be described by a regular expression

• Essential aspects of regular expressions can be specified with only
  
  0 or 1 The alphabet
  
  $R_1 R_2$ $R_1$ followed by $R_2$
  
  $R_1 + R_2$ $R_1$ or $R_2$ (same as `egrep’s |)
  
  $(R)$ Grouping
  
  $R^*$ Kleene closure: 0 or more $Rs$ $(10)^* (0+011+101+110)^* (01*01*01*)^*$

• What languages over \{0 1\} are regular? All but one below are regular

  Bit strings whose number of 0’s is a multiple of 5
  that begin with 0 and end with 1
  with more 1’s than 0’s
  with no consecutive 1’s
  for a binary number that is a multiple of 2
  for a binary number that is multiple of 5

• It is possible to cast *any* computation as a language problem
Finite State Automata

- A finite state automata, an FSA, is another representation for regular languages.
- A FSA is a simple machine with \( N \) states (0 to \( N-1 \))
  
  Start in state 0
  Read a bit
  Move to a new state depending on the bit and the current state
  Stop after reading last bit
  Accept if FSA is in one of its final states, Reject otherwise

- An FSA ‘recognizes’ its input: ‘Decides’ if the input is in the FSA’s regular language

10(10)* Transition table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- There is a one-to-one correspondence between FSAs and regular expressions.
- It is possible to construct FSAs automatically from regular expressions.

10101010? 00011110?
‘Bounce’ Filter

• Flip isolated 0s and 1s in a bitstream

  Input: 0 1 0 0 0 1 1 0 1 1
  Output: 0 0 0 0 0 1 1 1 1

• State interpretations
  1. At least two consecutive 0s
  2. Sequence of 0s followed by a single 1
  3. At least two consecutive 1s
  4. Sequence of 1s followed by a single 0

• Do ‘output’ by monitoring the state transitions
Simulating FSAs

```c
int main(int argc, char *argv[]) {
    int i = 0, zero[100], one[100], final[100];
    for (i = 0; i < 100; i++)
        if (scanf("%d%d%d", &zero[i], &one[i], &final[i]) != 3)
            break;
    for (i = 1; i < argc; i++) {
        int state = 0;
        char *input = argv[i];
        for (; *input != '\0'; input++)
            if (*input == '0')
                state = zero[state];
            else
                state = one[state];
        if (final[state])
            printf("%s: accepted\n", argv[i]);
        else
            printf("%s: rejected; ended in state %d\n", argv[i], state);    
    }
    return 0;
}
```

% cat fsainput
3 1 0
2 3 0
3 1 1
3 3 0

% lcc fsa.c
% a.out 10101010 10 101011 <fsainput
10101010: accepted
10: accepted
101011: rejected; ended in state 3
FSAs Can’t ‘Count’

• Theorem: No finite state machine can decide whether or not its input has the same number of 0s and 1s

• Proof

Suppose an $N$-state machine can determine if its input has equal number of 0s 1s
Give it $N+1$ 0s followed by $N+1$ 1s
Some state \textit{must} be visited a least twice
So, the machine would accept the same string \textit{without} the intervening 0s
And that string doesn’t have the same number of 0s and 1s. Contradiction \hfill \square

0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

• Need more powerful machines than FSAs

How much more powerful? Language hierarchy

Regular Finite-state automata
Context-free Pushdown automata (can count 2 things)
Context-sensitive Linear-bounded automata
Type 0 Turing machines

Take COS 487, Theory of Automata and Computation
Lecture 22. Hard Problems

• Important properties of algorithms
  
  **Finite:** Guaranteed to terminate
  
  **Deterministic:** Always produces the same output for the same input

• **Efficient** algorithms execute in times that are no more than *polynomial* in the size of their inputs, \( N \)
  
  \( N, N^2, N + N^4, \text{ etc.} \)

• **Inefficient** algorithms execute in times that are at least *exponential* in \( N \)
  
  \( 2^N, 10^N, N!, \text{ etc.} \)

• Some apparently simple problems have no known efficient solutions

  Traveling Salesman  Find the minimum-cost tour of \( N \) cities
  
  Scheduling  Schedule \( N \) jobs of varying length on two machines to finish by a given deadline
  
  Sequencing  Arrange \( N \) 4-letter fragments cut from a long string (with overlaps) into the original string (DNA sequencing)
  
  Satisfiability  Assign true/false values to \( N \) logical variables so that a given logical formula is true
The Traveling Skibum Problem

- Visit $N$ ski areas in the order that minimizes cost, e.g., distance

- To find an optimal tour, try all of them

```c
void visit(int k) {
    if (k == 1)
        checklength();
    else {
        for (i = 0; i < k; i++) {
            swap(i, k - 1);
            visit(k - 1);
            swap(i, k - 1);
        }
    }
}
```

- Takes $N!$ steps; no computer can run this for $N = 100$, because $100! \approx 10^{157}$

- Use *heuristics* to get good, but not optimal solutions, to hard problems
  
  TSP: Choose the ‘nearest neighbor’ as the next ski area on the tour

- Hard problems can be your friends: Use encryption to send secret messages
Unsolvable Problems

• Oh oh… Are some problems *unsolvable*?

• Example: Post’s Correspondence Problem

  \( N \) types of cards, each with a top string and a bottom string

  Using as many of each card as needed, arrange them so that the top and bottom strings are identical (or say it’s impossible)

  \[
  \begin{align*}
  \end{align*}
  \]

• There’s no solution for the cards

• The bad news: Post’s Correspondence Problem is unsolvable; you cannot write a program that determines if there is a solution for a given set of cards

\[
\begin{align*}
&AB AB AB AB AB AB A \\
&AB A B A B A B A B A \\
&A B A B A B A B A B A
\end{align*}
\]
The Halting Problem

• Write a C program that

  Reads another C program, \( P \)

  Reads \( P \)'s input

  Determines whether or not \( P \) loops forever; that is, whether or not \( P \) halts

  \[
  \text{while (x != 1)} \\
  \quad \text{if (x > 2) x -= 2; else x += 2;}
  \]

  \[
  \begin{array}{cccc}
  7 & 5 & 3 & 1 \\
  \end{array}
  \quad P \text{ halts}
  \]

  \[
  \begin{array}{cccccc}
  8 & 6 & 4 & 2 & 4 & 2 & 4 & \ldots \\
  \end{array}
  \quad P \text{ loops on even inputs}
  \]

  \[
  \text{while (x != 1)} \\
  \quad \text{if (x \% 2 != 0) x = 3*x + 1; else x /= 2;}
  \]

  \[
  \begin{array}{cccccccc}
  7 & 22 & 11 & 34 & 17 & 52 & 26 & 13 & 40 \\
  20 & 10 & 5 & 16 & 8 & 4 & 2 & 1 \\
  \end{array}
  \quad \text{does } P \text{ halt for all odd integers?}
  \]

  \[
  \begin{array}{cccc}
  8 & 4 & 2 & 1 \\
  \end{array}
  \quad P \text{ halts}
  \]
The Halting Problem, cont’d

• Theorem: The Halting Problem is unsolvable

• Proof by contradiction

Assume there is a program, $\text{HALTS}(P,y)$, that takes two inputs, a program $P$ and its input $y$. If $P(y)$ halts, $\text{HALTS}(P,y)$ stops and prints ‘Yes’; if $P(y)$ does not halt, $\text{HALTS}(P,y)$ stops and prints ‘No’

Build another program, $\text{CONFUSE}(x)$, that takes a legal C program $x$ as input. If $\text{HALTS}(x,x)$ prints ‘Yes’, $\text{CONFUSE}(x)$ loops forever; if $\text{HALTS}(x,x)$ prints ‘No’, $\text{CONFUSE}(x)$ stops.

Now, call $\text{CONFUSE}(\text{CONFUSE})$:

- If $\text{HALTS}(\text{CONFUSE},\text{CONFUSE})$ prints ‘Yes’, $\text{CONFUSE}(\text{CONFUSE})$ loops
- If $\text{HALTS}(\text{CONFUSE},\text{CONFUSE})$ prints ‘No’, $\text{CONFUSE}(\text{CONFUSE})$ stops

But $\text{CONFUSE}$ can’t do both! So, $\text{HALTS}$ cannot exist.

• Maybe C programs are too hard; what about TOY programs?
  If the Halting Problem can be solved for TOY programs, it can be solved for C
  Use a C compiler to translate C programs to TOY code

• Ditto for simple, abstract machines — for any machine that can simulate others
More Integers or Reals?

• Just how many unsolvable problems are there?

• A simpler question: Are there more integers or more even integers?

  0  1  2  3  4  5  6  7  8  9  10  11  12  ...
  0  2  4  6  8 10 12 14 16 18 20 22 24  ...

  There’s a 1-to-1 correspondence, none missing, so there are as many integers as
  even integers!

• Are there more integers or more reals? Try the same technique: Make a 1-to-1 correspondence between integers and reals, listing the reals in any order

  0  0.1001001100001001010010101...
  1  0.0001000100100101001000101...
  2  0.111111111111111111111111...
  3  0.00010000000100010001000010...
  4  0.1000000000000000000000000...
  5  0.11111101100011100011000111...

  This diagonalization shows there’s at least one real not on the list! 0.010011...
  the complement of the bits on the diagonal above

  There are infinitely more reals than integers

• All possible programs correspond to the integers, all possible functions correspond to the reals: Most functions are not computable!
Implications

• Practical
  Computing has its limitations; work within them
  Recognize and avoid unsolvable problems
  Recognize hard problems, don’t try for optimal solutions
  Use heuristics for hard problems
  Abstract structures reveal much about practical problems

• Philosophical (Buyer beware: Consult a ‘real’ philosopher for the truth)
  We ‘assume’ that step-by-step reasoning can solve any technical problem
  ‘Not quite’ says the Halting Problem
  Anything that is ‘like a computer’ suffers the same flaw
    Physical machines
    Human brain?
    Matter?
Lecture 23. Viruses and Secret Messages

• Remember `sum.toy`?

```
0E:  B001  R0 <- 01
0F:  B10A  R1 <- 0A
10:  B201  R2 <- 01
11:  B300  R3 <- 00
12:  2110  R1 <- R1 - R0
13:  6118  jump to 18 if R1 < 0
14:  1332  R3 <- R3 + R2
15:  1220  R2 <- R2 + R0
16:  2110  R1 <- R1 - R0
17:  5013  jump to 13
18:  4302  print R3
19:  0000  halt
```

starting address
R0 holds 1
R1 is n
R2 is i
R3 is sum
n--
if (n < 0) goto End
sum += i
i++
n--
goto Top
print sum

% /u/cs217/bin/toy /u/cs217/toy/sum.toy

0037

• Suppose an unknown source modifies `sum.toy` by appending the following code

```
87:  8088  R0 <- 88
88:  B108  R1 <- 08
89:  F201  R2 <- R0<<R1
8A:  C002  R0 <- R0^R2
8B:  4002  print R0
8C:  500E  jump to 0E
```

% /u/cs217/bin/toy /u/cs217/toy/sum.toy

8888

0037

`sum.toy` is infected with the ‘8888’ virus
Infection Routes

• If a virus $V$ can find a writable executable file $P$, it may be able to embed itself in $P$
  
  $\text{infect}(P,V)$  
  A copy of $P$ with $V$ embedded so $V$ gets initial control

  $V$'s execution can be arbitrarily complex, perhaps involving self-modifying code to cover its tracks

• When $\text{infect}(P,V)$ runs, $V$ can do anything $P$ can do, perhaps without visible effects
  
  Print ‘8888’
  
  Print

  $\text{login:}$

  On some other computer and wait for a user id; then print

  $\text{Password:}$

  Snarf the password entered, spawn another process running $/\text{bin/login}$, and leave town with a fresh user id and password; user just sees

  $\text{login:}$

  Scramble/delete your files

  Spawn a separate process running itself and find other executable files to infect
Detecting Viruses

• Given a program $P$, how can you tell if it’s infected? You can’t
• Virus detection software looks for occurrences of specific viruses
e.g.,
  Is the instruction at location $87_{16} = 8088_{16}$? ‘Infected with the 8888 virus’
  Oh oh… Viruses embed themselves in different ways and at different locations
  Must update virus detection software on a regular basis (daily?)
  Virus detection software does not solve the general problem ‘is $P$ infected?’
• Suppose you have two versions of supposedly the same program, $P_1$ and $P_2$
  Which one of $P_1$ or $P_2$ is infected?
  Do $P_1$ and $P_2$ produce the same output? (Even if one is infected)
  Both are unsolvable problems alà the Halting Problem
• Is there any hope?
  Intractable problems — those with only exponential-time algorithms — come to the rescue
Fingerprints

• Suppose that given a file $P$, $H(P)$ is a relatively small number that ‘characterizes’ $P$
  \[ H(\text{/u/cs126/examples/compile.c}) = 364BFFB1_{16} \]
  $H$ provides a fingerprint of /u/cs126/examples/compile.c
  Accept $P_2$, a copy of $P$, only if $H(P_2) = 364BFFB1_{16}$

• $H$ must be a one-way hash function with the following properties
  Given $P$, it must be easy to compute $H(P)$
  Given $H(P)$, it must be computationally infeasible to reconstruct $P$
  Given $P$ and a virus $V$, it must be computationally infeasible to arrange for $H(\text{infect}(P,V)) = H(P)$; that is, to find two bit strings with equal fingerprints

• Good one-way hash functions produce fingerprints with at least 128 bits
  \[ \text{MD5(compile.c)} \quad 979a7c5c \text{ ae9f12e2 702fc6ad 9ad4493a} \]
  \[ \text{SHA(compile.c)} \quad 85025ddc \text{ bb5c8da7 44598fe0 d8b5e16d a75cb560} \]
Fingerprints on the Internet

% ftp ftp.cs.princeton.edu
ftp> cd /pub/packages/cii
ftp> ls
README
cii10.tar.gz
cii10.tar.Z
cii10.zip
ftp> get README |more
...
The distribution directory contains the following files and directories. MD5 fingerprints for the files in this directory are listed below.
...
MD5 (cii10.tar.Z) = ba5b3c3b6c43061e4519c85f103be606
MD5 (cii10.tar.gz) = e3769ae75e52427e1b807e02aae3e
MD5 (cii10.zip) = fa71f475c97a4bfae66767012367c77f
ftp> get cii10.zip
ftp> quit
% md5 cii10.zip
MD5 (cii10.zip) = fa71f475c97a4bfae66767012367c77f

- This isn’t foolproof — intruders can intercept Internet packets and substitute different fingerprints
Cryptography

- A *cryptosystem* keeps secret messages (and files) from prying eyes

```
void encrypt(char *buf, int len, char *key, int keylen) {
    int i = 0;
    for (i = 0; len-- > 0; i = (i + 1)%keylen)
        *buf++ ^= key[i];
}
```

- Modern cryptosystems exclusive-OR key with plaintext: \( C = P \oplus K \)

Watch out! Sending many 0s in plaintext gives attackers pure key: \( C = 0 \oplus K = K \)
Cryptography, cont’d

• Repeated use of a relatively short key isn’t secure; most systems use the key to generate a long stream of pseudo-key, which is XOR’d with the plaintext

• Assume the worst: Attackers know the algorithm, the length of the key, and have the ciphertext

• Security rests on the strength of the algorithm and the security of the key

• Best systems force attackers to use inefficient algorithms, which require trying try all $2^n$ n-bit keys; just use large $n$

• Designing secure cryptosystems sounds easy, but it’s not; don’t trust amateurs!

• Key distribution is just as hard as encryption: What’s the best way to exchange keys with your trusted correspondents and keep them secret? There isn’t one…

Public-Key Cryptosystems

• **Public-key** cryptosystems avoid the key distribution problem by using *two keys*

  Everyone knows your public key, \( P \)
  
  Only you know your secret key, \( S \)
  
  To send \( M \): Send \( P_{drh}(M) \) via any medium
  
  To read \( M \): I read \( S_{drh}(M) \)

• List public keys in the phone book, or its equivalent

  % finger -l drh@cs.princeton.edu
  ...
  -----BEGIN PGP PUBLIC KEY BLOCK-----
  Version: 2.6.1
  mQBNAiluT8gAAAECAK8TOxmBQ6XhoJXrGPtDKzhZkIqSRh3pMimt8nUh1nSfByec
  KittyH02STppLwncD47j8KK6Cm5hriyzusnX/hkABRG0JkRhdmlkIFIuIEhhbnNv
  biA8ZHJoQGNzLnByaW5jZXVibi5lZHU+
  =JFCd
  -----END PGP PUBLIC KEY BLOCK-----

• For all public-key algorithms

  \[ S(P(M)) = M \text{ for all } M \]
  
  All \( S, P \) pairs must be distinct
  
  Deriving \( S \) from \( P \) must be as hard as reading \( M \)
  
  \( P(M) \) and \( S(M) \) must be efficient
RSA Public-Key Cryptosystem

- The RSA cryptosystem uses arithmetic on very large integers
  - \( P \) is \( N, p \)
  - \( S \) is \( N, s \) where \( N \approx 200 \) digits, \( p \) and \( s \approx 100 \) digits
- To choose \( N, p, s \)
  - Pick 3 100-digit secret prime numbers, \( x, y, s \)
  - The largest is \( s \)
    - \( N = x \times y \)
    - \( N = 47 \times 79 = 3713 \)
  - Choose \( p \) so that \((p \times s) \mod ((x - 1)(y - 1)) = 1\)
    - \( p \times 97 \mod (46 \times 78) = 1 \)
    - \( 37 \times 97 \mod 3588 = 1 \)
    - \( 3589/3588 = 1 \) remainder 1
- Attackers see only \( N \) and \( p \)
  - To find \( s \), attackers must factor \( N \) into its prime factors \( x \) and \( y \)
    - It is believed, but not proven, to be infeasible to factor \( N \) if it’s sufficiently large
    - Factoring 200-digit numbers probably takes \( \approx 10^9 \) years
- Are there enough primes for everyone? Yes: \( \approx 10^{150} \) primes with \( \leq 512 \) bits (\( \approx 155 \) decimal digits)
RSA Encryption

- **To encrypt** $M$, use $N$ and the **public** key, $p$

  Encode $M$ in numbers $< N$

  For each $M_i$, $C_i = M_i^p \mod N$  
  
  the remainder of $M_i^p$ when divided by $N$

For $N = 3713$, $p = 37$, $s = 97$

$M$  

Please send money

```
Please _send_ money_
```

Encode:  

```
1612 0501 1905 0019 0514 0400 1315 1405 2500
```

Encrypt:  

```
2080 0057 1857 3706 1584 0888 2067 0591 1277
```

$1612^{37} = 47,044,232,358,938,497,020,498,996,761,564,680,247,331,818, 
462,325,046,870,527,453,082,869,350,611,474,961,064,423,374, 
436,277,844,788,137,937,637,623,201,792$

$1612^{37} \mod 3713 = 2080$, etc.
RSA Decryption

• To decrypt $M$, use $N$ and the private key, $s$
  
  For each $C_i$, $M_i = C_i^s \mod N$

  Decode numbers to reveal $M$

  For $N = 3713$, $p = 37$, $s = 97$

  Please send money

  $C$: 2080 0057 1857 3706 1584 0888 2067 0591 1277

  Decrypt: 1612 0501 1905 0019 0514 0400 1315 1405 2500

  $57^{97} = 208,862,754,025,291,103,893,549,722,030,506,307,840,035,159,$
  $185,066,358,136,864,739,390,751,752,973,213,714,581,100,145,$
  $330,888,003,488,562,198,990,224,718,358,613,240,589,340,493,287,$
  $521,060,551,858,632,460,253,869,992,608,057$

  $57^{97} \mod 3713 = 501$

  Decode: 1612 0501 1905 0019 0514 0400 1315 1405 2500

    _PLEASE SEND MONEY_

• This example is from R. Sedgewick, *Algorithms in C*, Addison-Wesley, 1990

• For details on multiple-precision arithmetic, see D. R. Hanson, *C Interfaces and Implementations*, Addison-Wesley, 1997
PGP

- PGP — Pretty Good Privacy — is widely used public-key cryptosystem available for PCs, UNIX systems, etc.

```
you% cat / pgp -fea drh
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
Can I have more time on the current programming assignment?
--frazzled in Princeton
^D
-----BEGIN PGP MESSAGE-----
Version: 2.6.2
hEwDriyzusnX/hkBAGChqSkxFkFwyMFyCwrc187jHzXshOdrDQYTDQbRwwVcGZIyA83TTPYzFGU3yHHnNVQHAEjJDRJRHPaEXRNEUippgAAAGjcN7B2zmqgvJeW1iR2
dTOVQtmsN9Ez32CdYD8ub/3b7sm8xq+NCBm13/83TexSgyudPaqPoifd7q0N96z
kL4tSAmcJHwfzyiM/RJ+2p41YgcgAqFgaB2NTHaowXQpG4qNg3nMSTxOg==
=5u0S
-----END PGP MESSAGE-----
```

```
you% cat / pgp -fea drh / mail drh@cs

drh% inc
Incorporating new mail into inbox...
  92+ 09/04 To:drh@fs.CS.Prin <<-----BEGIN PGP MESSAGE-----
drh% show / pgp -fd
Can I have more time on the current programming assignment?
--frazzled in Princeton
```